Master's Thesis

Czech
Technical
University
in Prague

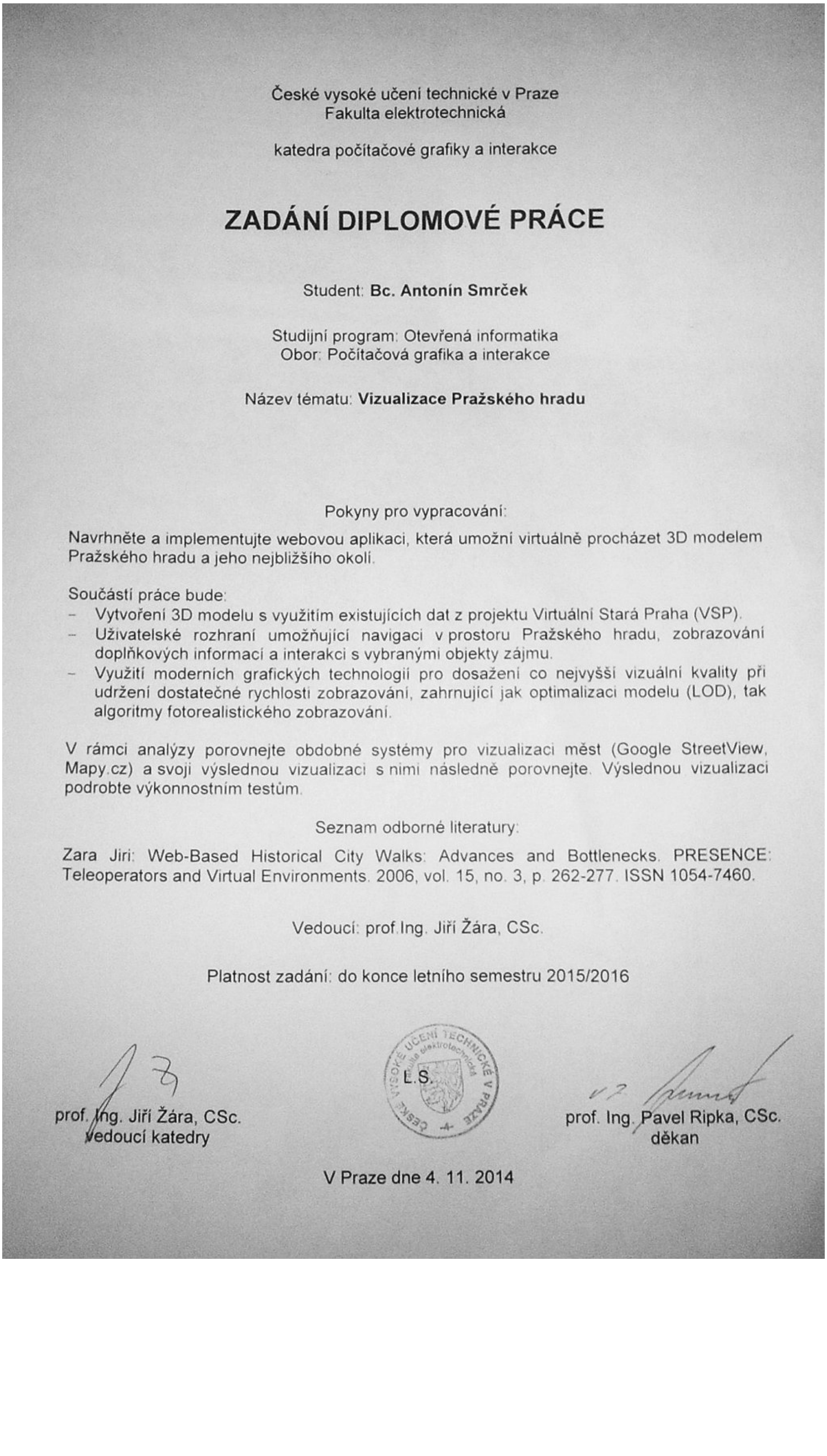
F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Visualization of Prague Castle

Bc. Antonín Smrček
Study Programme: Open Informatics
Field of Study: Computer Graphics and Interaction

January 2016
Supervisor: prof. Ing. Jiří Žára, CSc.



České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Antonín Smrček**

Studijní program: Otevřená informatika
Obor: Počítačová grafika a interakce

Název tématu: **Vizualizace Pražského hradu**

Pokyny pro vypracování:

Navrhněte a implementujte webovou aplikaci, která umožní virtuálně procházet 3D modelem Pražského hradu a jeho nejbližšího okolí.

Součástí práce bude:

- Vytvoření 3D modelu s využitím existujících dat z projektu Virtuální Stará Praha (VSP).
- Uživatelské rozhraní umožňující navigaci v prostoru Pražského hradu, zobrazování doplňkových informací a interakci s vybranými objekty zájmu.
- Využití moderních grafických technologií pro dosažení co nejvyšší vizuální kvality při udržení dostatečné rychlosti zobrazování, zahrnující jak optimalizaci modelu (LOD), tak algoritmy fotorealistického zobrazování.

V rámci analýzy porovnejte obdobné systémy pro vizualizaci měst (Google StreetView, Mapy.cz) a svoji výslednou vizualizaci s nimi následně porovnejte. Výslednou vizualizaci podrobte výkonnostním testům.

Seznam odborné literatury:

Zara Jiri: Web-Based Historical City Walks: Advances and Bottlenecks. PRESENCE: Teleoperators and Virtual Environments. 2006, vol. 15, no. 3, p. 262-277. ISSN 1054-7460.

Vedoucí: prof.Ing. Jiří Žára, CSc.

Platnost zadání: do konce letního semestru 2015/2016

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

v.z.
prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4. 11. 2014

Acknowledgement

I would like to thank prof. Ing. Jiří Žára, CSc., for supervision of my thesis. He provided me with invaluable insights, feedback on work in progress and his guidance was especially important in finishing stages of the project. My thanks are also due to all friends who helped me with the user testing. Finally, I want to particularly thank my mother for her support throughout my studies.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on January 11, 2016

. . . .

Abstrakt / Abstract

Tato diplomová práce se zabývá vývojem webové aplikace která vizualizuje prostředí Pražského hradu a umožňuje 3D virtuální procházku v rámci jeho prostor. Důraz je kladen na grafickou kvalitu a výkon aplikace. Obdobné systémy pro vizualizaci měst jsou podrobeny analýze a jsou diskutovány možné volby technologií pro implementaci. Je vysvětlen proces návrhu a implementace uživatelského rozhraní, stejně tak jako zvolený přístup ohledně modelování, začlenění moderních vizuálních efektů a dosažení výkonnostních optimalizací. Finální aplikace je podrobena uživatelskému a výkonnostnímu testování a je provedeno srovnání s obdobnými vizualizačními aplikacemi.

Klíčová slova: Pražský hrad, webová vizualizační aplikace, virtuální procházka, WebGL, Three.js

Překlad titulu: Vizualizace Pražského hradu

This thesis describes a development of the web-based 3D virtual walk application which visualizes the Prague Castle and provides the users with an information about interesting objects in the area, with focus on graphical quality and performance of the application. Similar city visualization applications are analyzed and choice of technologies used for the implementation is discussed. Process of the user interface design and implementation is explained, as well as the approach taken in modeling, incorporating modern visual effects and achieving performance optimizations. We have performed user testing and also measured the application's performance. Finally, we compare the application to other visualization applications.

Keywords: Prague Castle, web-based visualization, virtual walk, WebGL, Three.js

Contents /

Tables / Figures

Chapter 1
Introduction

With the rise of computers' performance and especially their 3D capabilities which has happened over the last two decades, it became possible to create virtual "copies" of the real world places and even walk around them. One of the notable projects from the past which focused on the theme of virtual walks is the Virtual Old Prague project. We consider it as the predecessor of our application, having the same theme and even sharing part of visualized areas – the Prague Castle in this case. The Virtual Old Prague project is originally from 1999 and times have changed since then. Internet connection speed is significantly faster, performance of average PC has increased several-fold. The things which were once crucial for realizing such application are not anymore, but there are new things to consider. Web browser vendors have been gradually removing browser plugin support in favor of standards-based web technologies, making some of the popular technology choices for virtual walks such as VRML practically unusable; on the other hand, there are much more technologies to choose from nowadays. The web and 3D content generally is being accessed from more types of devices apart from desktop computers – laptops, mobile phones, tablets etc., making it possible to broaden the audience even further.

The goal of our project is to develop a modern web-based 3D virtual walk application which visualizes the Prague Castle and provides users with information about interesting objects in the area, with focus on graphical quality and performance. The aim of the thesis is to provide insight into the area of virtual walk applications in the context of present-day technologies. The current state of virtual walk applications is explored and desirable traits are collected, and we show how we use this knowledge in design and implementation of user interface for our application. We present the reader with our approach to creation of the virtual world (modeling, physics simulation, visual effects) and means used to achieve desired performance, including specific implementation details and explanation of reasoning behind. To verify performance of the application and ensure its practical usability by regular users, we have also conducted performance and user testing. It is also shown how our application fares in visual comparison with similar visualization applications and, what is probably the most interesting comparison, how much the technology has evolved over the last years by comparing same areas from our application and Virtual Old Prague project against each other.

1.1 Thesis structure

The thesis is structured as follows: **Chapter 2** deals with an analysis of related similar visualization systems. **Chapter 3** addresses pros and cons of technologies available for a development of web-based 3D virtual walk application. The project specification is given in **Chapter 4**, which focuses in closer detail on the particular goals and decisions concerning the implementation and used technologies. Structure of the developed virtual walk application is presented in **Chapter 5**. In **Chapter 6**, the process of 3D content creation for the application is discussed. **Chapter 7** is devoted to design and implementation of the user interface. Implementation specifics are described in **Chapter 8** (collision detection) and **Chapter 9** (performance optimizations), and **Chapter 10** talks about the lighting and choice of visual effects used to improve the visual quality of the application. The results of the user and performance testing are presented in the **Chapter 11**. Finally, **Chapter 12** concludes the thesis and outlines the possible future work.

Chapter 2
Visualization systems analysis

In this chapter, we will introduce a number of existing web-based city visualization systems (divided into the two main categories – **map services** and **standalone presentations**), discuss their functionality with relation to virtual walks and compare them against each other based on selected evaluation criteria.

2.1 Goals

Goal of our analysis is to gain an insight into the state of visualization applications with a similar theme and collect desirable traits for a good virtual walk. We aim to analyze applications' strengths and weaknesses and use this information during the development of our own application. The analysis consists of a text review and rating of specified qualities. A quality can be rated using these marks (signs):

- negative (-)
  - quality is *missing* or on a *poor* level
  - in case of negative qualities (i.e. "missing models"), it means that the application contains this negative quality (e.g. some models in the application are missing)
- zero/neutral (0)
  - quality is considered *acceptable* (neither bad, nor very good)
- positive (+)
  - quality is *present* (not missing) or *very good*
  - in case of negative qualities (i.e. "missing models"), it means that the application does not possess this negative quality (e.g. it has no missing models)

Each quality has associated weight between 0 and 1 which signalizes its importance. Sum of weights of all qualities is 1. Rated qualities are different for map services and standalone presentations – specification is given at the beginning of associated sections. In the end of the analysis, ratings of the applications are presented. Total rating for an application is computed as a sum of *additions* from all rated qualities and this sum is mapped from its [-1,1] range to the 0-100 % range. Addition of a quality is computed as:

$addition = sign(givenMark) * weight$

The applications are compared against each other according to their ratings. The conclusion of our analysis is drawn based on the results of this comparison.

2.2 Map services

Map services are applications where **city visualization is not the sole purpose**. They provide wide range of services for users such as browsing of various 2D map types, route planning, displaying public transport timetable in places of interest and so on. City visualization is present in a form of panoramic views from positions along city streets, and in recent years, also in a form of 3D models created computationally from an aerial imagery. Four popular map services are introduced and compared – **Google Maps**, **Google Earth**, **Mapy.cz**, and **Here**.

The rated qualities for map services were chosen as following:

- Model quality
  - geometry – correct shape of objects without artefacts (holes, bends, ...)
  - textures – visual quality (resolution, image deformations, ...)
- Missing objects
  - Are all objects 3D models? (some of them can be just photographs etc.)
- Fully 3D view
  - Is it possible to freely manipulate the view? (arbitrary angle and position)
- Special features
  - notable unique features, e.g. an ability to perform an actual virtual walk through a city
- User Interface
  - whether and how it provides the information about objects of interest, navigation in the area and a way how to control the avatar/manipulate the application
  - well-arrangement, simplicity, visual appeal

2.2.1 Google Maps

Available at: `https://www.google.com/maps`

Google Maps is a web mapping service provided by Google since 2005. It offers classic and satellite maps, as well as an option of functional layers above the maps, for example visualization of the current area traffic. Satellite maps are often just 2D maps, but in selected cities around the world[1]), sort of 3D view is present, which displays layer containing 3D models of buildings (using WebGL) and lets user to manipulate with the view.

The manipulation in this 3D view mode is severely limited; user can tilt the view only in three fixed positions:

- Orthogonal
- Bird's eye view
- Almost parallel to the ground

Orthogonal view is the default one where the user is looking to the ground right from the above. The experience is almost the same as with a normal 2D satellite map – 3D models are not recognizable from such angle.

[1]) Cities mainly in USA and Europe, full list at `http://support.google.com/maps/answer/2789536?hl=en&ref_topic=6002888`

Probably the most interesting is the **bird's eye view** from 45 ° angle, where a small to medium portion of a city is viewed with a sense of plasticity from now clearly recognizable 3D models. City structure is easy to see – this view is similar to the view from a sightseeing flight above the city. Last view option is almost **parallel to the ground** (approaching 90 °). Apart from that, it is similar to the previous and allows user to see the whole (or at least very large) part of a city and its wide surroundings. Zoom can be manipulated freely (although there is a limit of maximal zoom), but details on the maximum zoom are very poor and the view is quite confusing, because satellite photographs and 3D models are mixed together. For example, roads have photographs containing objects (like cars) as textures; the objects of course do not appear 3D, instead they appear "flat", as if lying on the road. Rotation cannot be manipulated freely and is limited (similarly to tilting) only to four positions, which are 90 ° apart (i.e. North, East, South, West). Despite the fact that the view manipulation cannot be fully controlled, 3D models in Google Maps provide a decent way of visualizing a city structure and if they are not examined from a close distance, they look impressive.

Lets move further to the another mode of Google Maps called the **Street View** – this popular mode with panoramic photographs is probably closest to the idea of a virtual walk. But no matter how clever the idea behind panoramic photographs is, it can never be a virtual walk in its "true" sense, because it is still just photographs and user cannot move freely in all three dimensions of the world.

As for the user interface (see Figure 2.1[1]), Google really uses its biggest trump card of interconnection with the whole Google ecosystem (search engine and other applications) to the maximum. On the left part of the screen, the information about objects of interest is displayed, with a possibility to browse photographs or read reviews written by the users. The photographs are also displayed in a form of a list in the lower part of the screen. The icon for displaying the classical map has reserved small space on the left of this list. Objects of interest are marked in the 3D virtual world (and classical map) or can be searched in the database using the search bar above the information section. The right lower part of the screen contains the control panel with tilting, rotation and zoom controls plus a yellow icon of a sticky figure which can be placed into the map and transition the mode into the Street View. All information and photographs are presented in a concise (but sufficient) manner, the whole user interface is very well-arranged, simple to use and nice to look at.

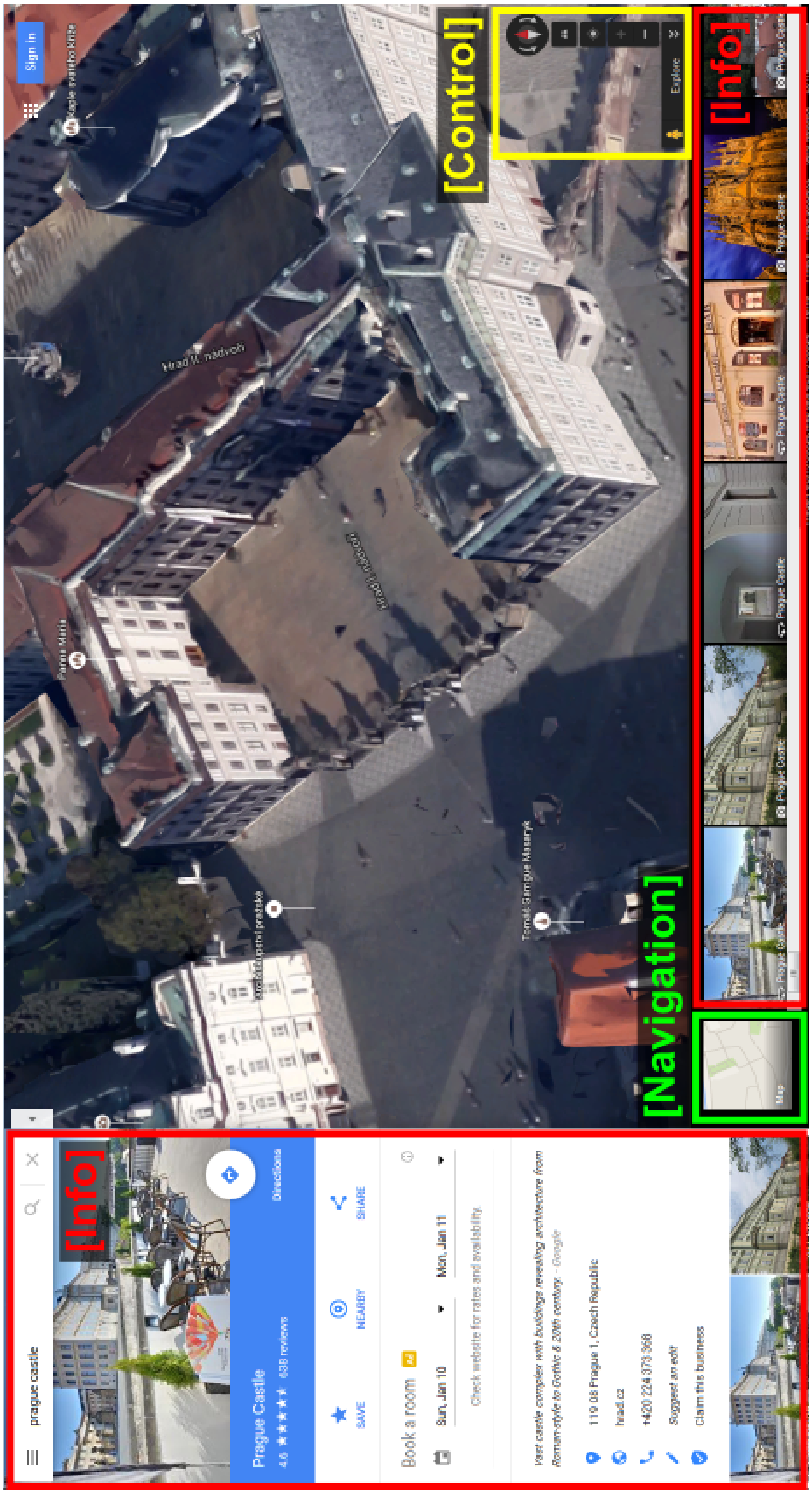
2.2.2 Google Earth

Google Earth is a geographical information program, virtual globe and map, currently developed by Google. It is free for personal use, commercial license exists too. User can view almost entire Earth in 3D, since it uses digital elevation model data collected by NASA [1]. The graphics is rendered either by OpenGL or DirectX.

Google Earth requires internet connection and exists as a standalone application in both desktop (Windows, OS X, Linux) and mobile versions (Android, iOS). First version was released in 2005 and has gained big popularity since, reaching over one billion downloads six years later [2]. In the early years of Google Earth, there were limited means to explore cities in 3D – only a few ones from US contained 3D models, and those were just gray blocky buildings. Google gradually improved quality of the models and also allowed contribution from users via SketchUp[2]) modeling application, so with

[1]) All Figures in this section contain captions enclosed in [brackets] which point out important parts of the user interface and possibly some other information – i.e. navigation within the world, information about objects of interest, controlling the application

[2]) `http://www.sketchup.com`



Prague Castle
4.6 ★★★★★ 638 reviews

Book a room Ad
Sun, Jan 10
Mon, Jan 11
Check website for rates and availability.

Vast castle complex with buildings revealing architecture from Roman-style to Gothic & 20th century. - Google

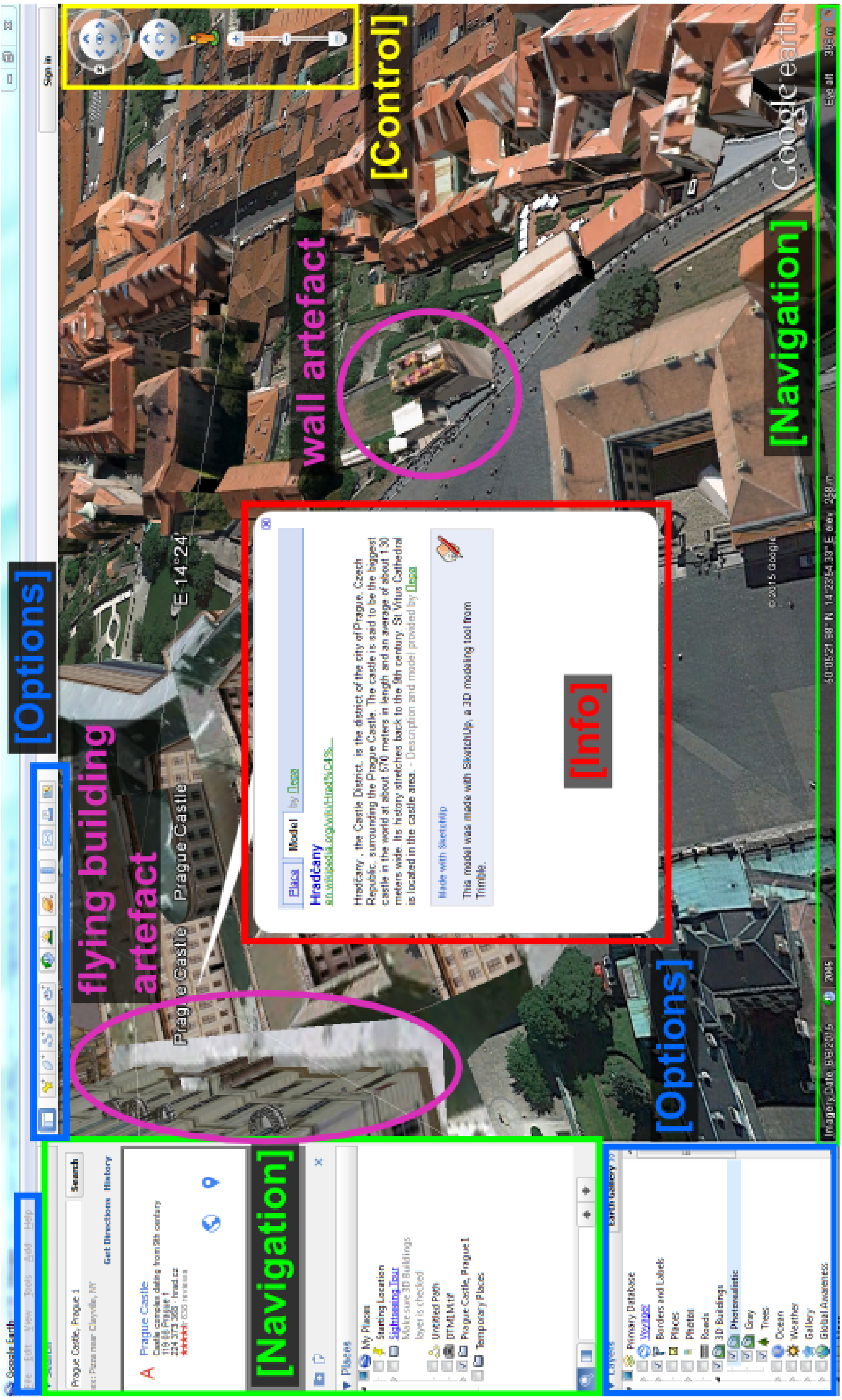
119 08 Prague 1, Czech Republic
hrad.cz
+420 224 373 368
Suggest an edit
Claim this business

Figure 2.1. Google Maps 3D (bird's eye) – Prague Castle

a help of the community, new building models quickly appeared all around the world. Despite that, there are still many buildings with no 3D models and only their satellite photographs are present at such places. In 2013, Google discontinued user model contributions [3], focusing on automatic creation using method of *Stereo Photogrammetry* [4–5], where multiple photographs of same objects are taken at different perspectives. Aircrafts are used for this task, making multiple flights over an area and capturing high resolution photographs.

Google Earth allows user to freely zoom and rotate around the 3D building models, and even a virtual walk from the first-person view is possible. However, there is no collision detection with buildings and the avatar often walks "under the ground" during walking around more complicated building structures (see Figure 2.3), because 3D models are in fact just a layer placed over the main 3D world and model's ground levels usually do not match those of the main world. Google Earth allows the user to find a lot more information about the area than Google Maps (more media, more text, more geographical-related information, more everything), which is probably why the user interface may be a bit complex sometimes. The arrangement of the user interface is similar to Google Maps – there is a control panel in the right part of the screen which allows manipulation with the zoom (unlimited), with the view and a sticky figure for entering the Street View mode. The information is not presented in such a compact way like in Google Maps, and objects of interest are not marked until the right option is turned on in the options. Only experienced users would be able to comfortably use the application.

The Prague Castle in Google Earth has still most of its models created by users and only a few not so significant buildings were created automatically, and the results are weird occasionally – see the textured blocks that are supposed to be castle walls on Figure 2.2, highlighted in pink. We have also experienced that some buildings are "flying" (again see Figure 2.2, highlighted in pink), which really does not make the best impression. Some buildings do not have any models available and only their satellite photograph is present. Because of the manual creation, buildings and surroundings have nice details (e.g. gates, fountains), their models have precise geometric structure that is very similar to the real one, and without artefacts from which automatically generated models suffer, such as edgy walls or collapsed parts. The big downside are textures, which are too much of a low quality for a visually appealing virtual walk – not to mention ground and house roofs textures, which are obviously from satellite images and of extremely low quality. On the bright side, looking at the castle from a distance hides these flaws, providing a rather accurate and nice view.



Place | Model by Dapa

Hradčany

en.wikipedia.org/wiki/Hrad%C4%...

Hradčany , the Castle District, is the district of the city of Prague, Czech Republic, surrounding the Prague Castle. The castle is said to be the biggest castle in the world at about 570 meters in length and an average of about 130 meters wide. Its history stretches back to the 9th century. St Vitus Cathedral is located in the castle area. - Description and model provided by Dapa

Made with SketchUp

This model was made with SketchUp, a 3D modeling tool from Trimble.

Figure 2.2. Google Earth – Prague Castle

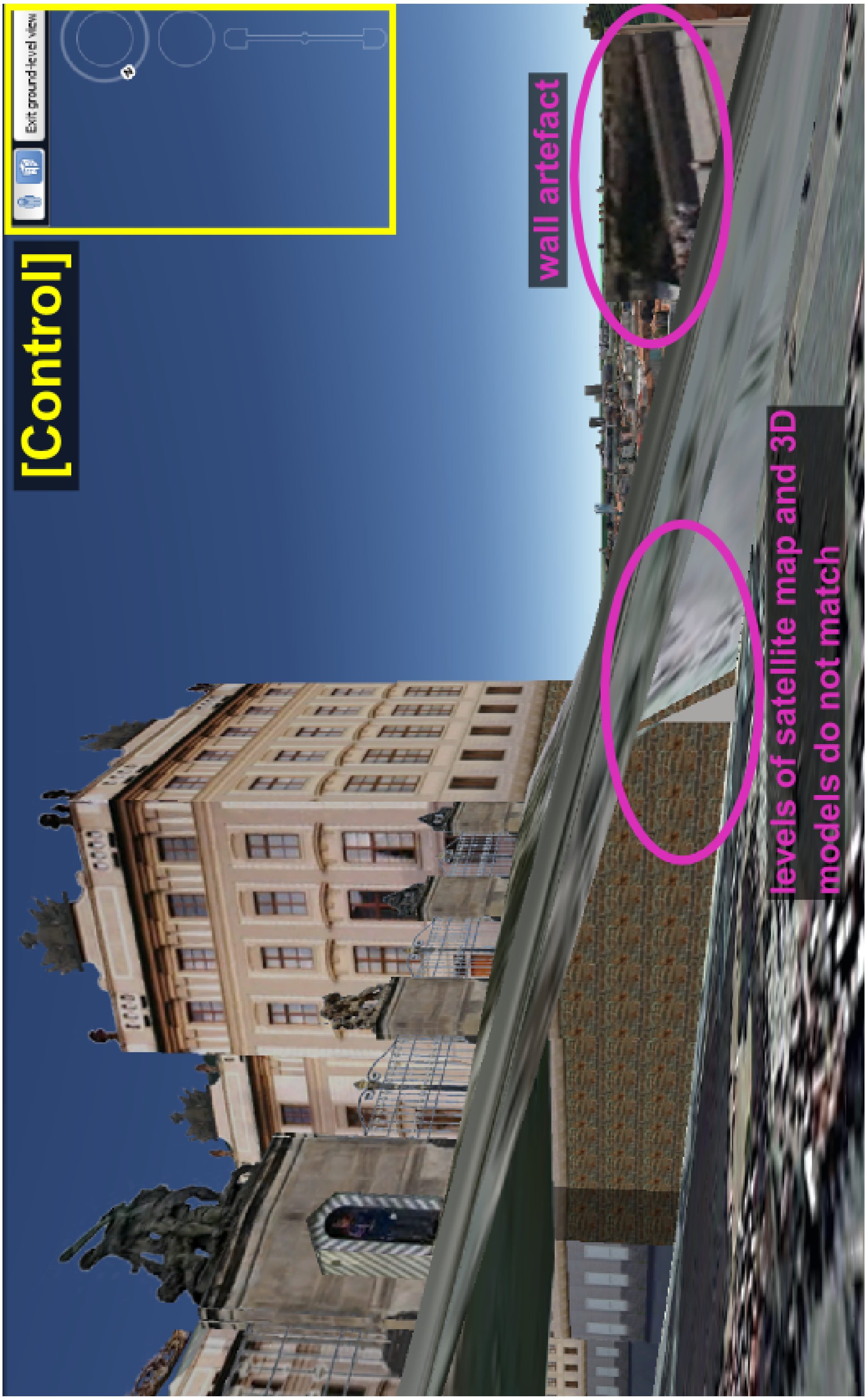


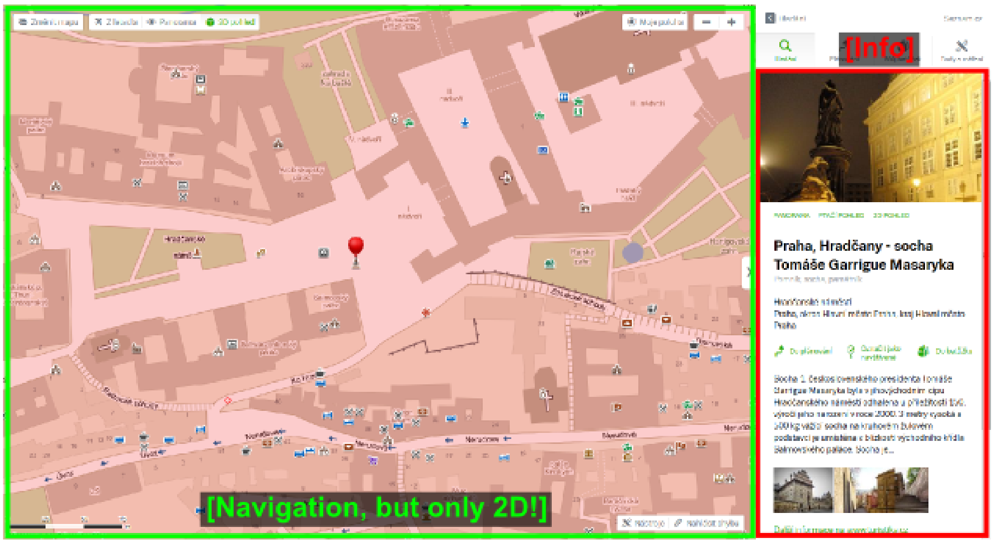
Figure 2.3. Google Earth – virtual walk in the Prague Castle

2.2.3 Mapy.cz

Available at: `http://mapy.cz`

Mapy.cz is a project focused mainly on the Czech Republic, developed by Czech company Seznam. It has its own version of Google's Street View called *Panorama*, which works exactly the same. In 2014, Seznam launched *3D map project* – 3D models of a few cities were created (e.g. part of Prague, Brno, Kutná Hora) and became available to explore. 3D maps run on WebGL technology and allow arbitrary rotation of a city model to the side and view from desired height, along with an option of *sight-seeing flight*, which makes the view rotate around the screen center. The user is allowed to zoom freely, but the maximal zoom is limited.

The whole project is powered by Melown Maps[1]) – a software developed by Citationtech, Czech technology company focused on the applied research and development in the field of computer vision. Melown Maps is a software system for creating 3D models of landscape from aerial photographs. The input of the system are uncalibrated aerial photographs, the output is a georeferenced 3D model. Data are processed automatically, without manual assistance [6]. Automatic computation is the reason why data processing takes a very long time – several years in general [7]. However, Vysoká Škola Báňská[2]) has also participated in the project, lending power of their "supercomputer" in order to speed up the computation process and cutting its length to one year. Seznam aims to gradually add models for more than 250 towns and cities in the near future.



Praha, Hradčany - socha Tomáše Garrigue Masaryka

Hradčanské náměstí
Praha, okres Hlavní město Praha, kraj Hlavní město Praha

Socha 1. československého presidenta Tomáše Garrigue Masaryka byla v jihovýchodním cípu Hradčanského náměstí odhalena u příležitosti 150. výročí jeho narození v roce 2000. 3 metry vysoká a 500 kg vážící socha na kruhovém žulovém podstavci je umístěna v blízkosti východního křídla Salmovského paláce. Socha je...

Další informace na www.turistika.cz

Figure 2.5. Mapy.cz – information about an object of interest is not visible in 3D mode

The quality of models is, as expected, not perfect. Buildings have usually more or less distorted shape. The impression is very good from further distance where details are not apparent, but the biggest zoom reveals a lot of imperfections. 3D maps are great for an overall feeling of the area and a general idea how the place looks like. The user interface (Figure 2.4) is simple and well-arranged. The lower part of the screen contains the control panel where it is possible to manipulate with the zoom, rotate/tilt the view or turn on/off the "sight-seeing flight". The left part of the screen is reserved for the display of a classical map where the current position in the 3D world is shown,

[1]) `http://www.melown.com/maps`

[2]) `http://www.vsb.cz/cs`

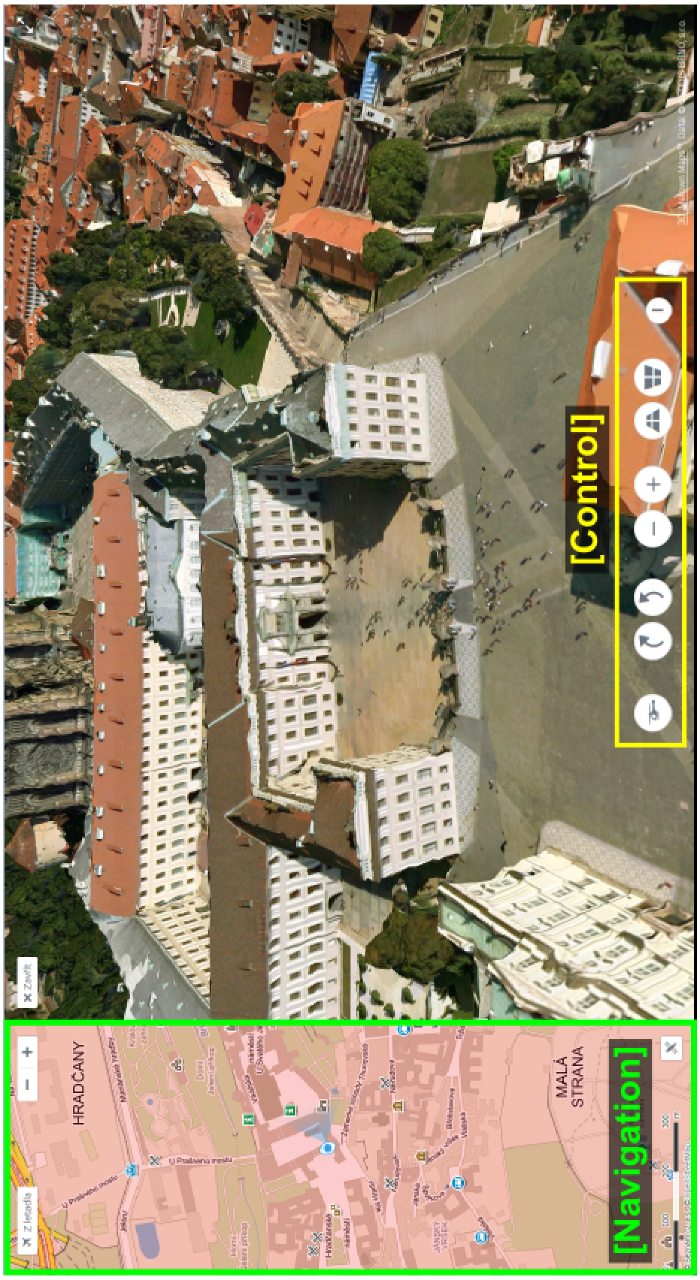


Figure 2.4. Mapy.cz – Prague Castle

along with objects of interest. However, the execution is not handled very well – when an object of interest is clicked, information window similar to Google Maps is displayed, but the whole view into the 3D world disappears (Figure 2.5). Also, unlike in Google Maps and Google Earth, objects of interest are not marked in the 3D virtual world.

2.2.4 Here

Available at[1]): `https://www.here.com`

Here is a mapping service developed by Nokia[2]), formerly known under the name Ovi Maps/Nokia Maps. It has very similar functions to Google Maps and runs in WebGL.

[1]) 3D map function no longer available since the beginning of 2015

[2]) `http://company.nokia.com/en`

Interesting feature is *Maps 3D*, launched in 2011 [8], which is extremely similar to those of Mapy.cz, but contains 3D models for much more cities, mainly located in Europe. The user interface (Figure 2.6) is pretty similar to Google Maps – info about objects of interest appear in the left part of the screen when opened, objects of interests or places can be searched in the database using the search bar in the top part of the screen (however, just like in Mapy.cz, objects of interest are not marked in 3D), and the lower part of the screen contains control elements for moving/tilting the view and (limited) zoom. Upper right part of the screen contains panel with various options, e.g. switching between map layers or teleporting to other cities.

2.2.5 Comparison of map services

The ratings for each application and their overall rating can be seen on Table 2.1. **Google Maps**' biggest disadvantage is absence of a full 3D view, which makes it really suitable only for obtaining general feeling of a place. Otherwise, it has acceptable virtual world qualities, especially quality of geometry is good. However, what shines the most is the simple, well-arranged and visually appealing user interface which provides enough information about the area and blends with the 3D view really well. **Google Earth** shares a lot of good qualities with Google Maps, but in contrast, it contains a full 3D view and even allows a virtual walk through a city; unfortunately this functionality suffers from technology flaws, but the experience is enjoyable nevertheless. Sadly, what brings overall feeling down is an absence of objects in certain areas, various artefacts and the user interface, which is rather complicated and information about objects of interest is not easily accessible. **Mapy.cz** and **Here** are very similar – Here models have less geometry anomalies, but Mapy.cz has better textures. The user interface of Mapy.cz is simple, 3D controls are quick to understand and easy to manipulate with; but the interaction with objects of interest is not executed very well. Here has less appealing controls, but the interaction with objects of interest is handled similar to Google Maps, which is good.

Weight	Criterion	Google Maps	Google Earth	Mapy.cz	Here
0.25	Model quality	0	0	0	0
0.05	Missing objects	0	-	0	0
0.20	Fully 3D view	-	+	+	+
0.05	Special features	0	+	0	0
0.45	User Interface	+	-	0	0
	TOTAL SCORE	**63 %**	**35 %**	**60 %**	**60 %**

Table 2.1. Comparison of map services

Based on the rated qualities, both Mapy.cz and Here placed on the same level, only a little behind Google Maps, which has placed first – although it lacks a full 3D view, it contains unparalleled user interface and interaction with objects of interest. The last place belongs to Google Earth due to the overly complex user interface. Visual comparison[1]) of Google Earth, Mapy.cz and Here is depicted on Figure 2.7 – as you can see, the overall visual quality is similar and indistinguishable from a distance.

[1]) We have omitted Google Maps from the comparison as it does not contain full 3D view and we would be unable to take the screenshot of the desired view where full castle area is clearly visible

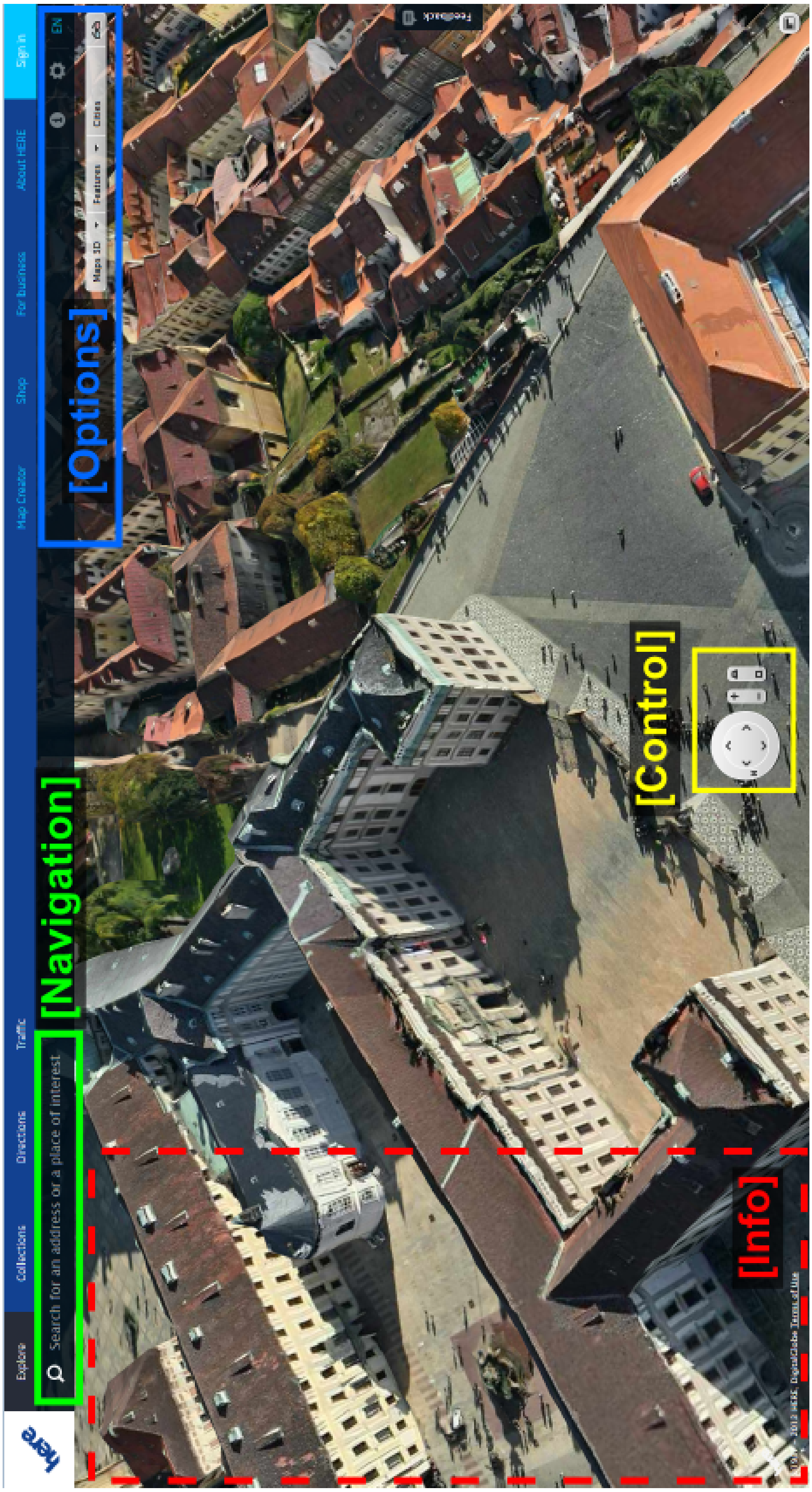


Figure 2.6. Here – Prague Castle

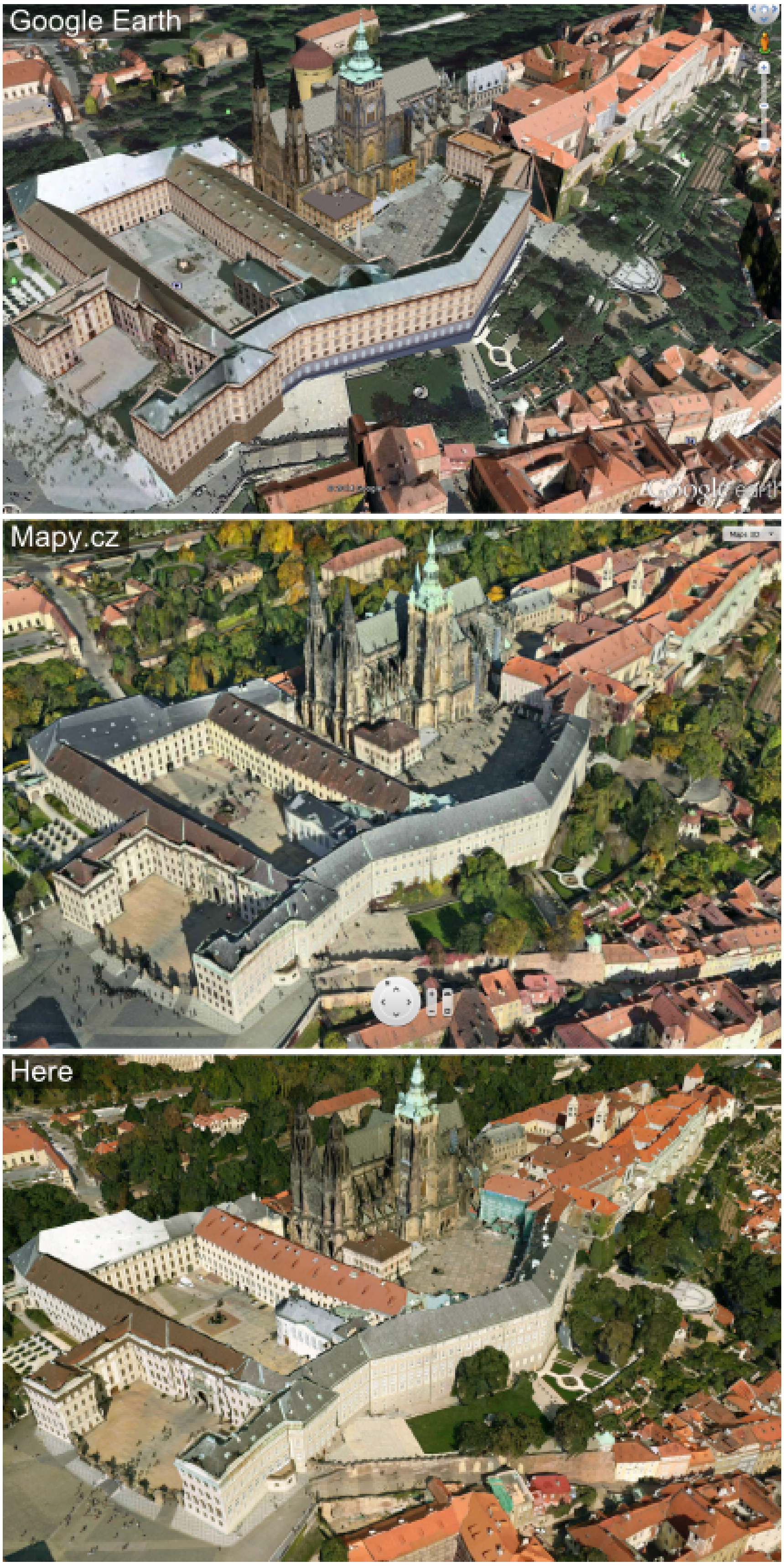


Figure 2.7. Visual comparison of map services with a full 3D view functionality

2.3 Standalone presentations

Standalone presentations focus on a very limited part of an area and **visualization is the major purpose**. Visualized areas are very often places with some historical value – both indoor (e.g. temples, chapels) and outdoor (e.g. squares, gardens), which are sometimes even combined into one 3D virtual walk. Because of a limited spatial range of a presented world, visualization can be more detailed and have better graphics quality. User interactivity is usually higher and application lets the user to explore the visualization freely, usually from the first-person view perspective.

There are very few standalone presentations which focus on a "true" city visualization. The most suitable representative presented is, without a doubt, the **Virtual Old Prague**, since it was the biggest inspiration to Visualization of Prague Castle and, in a sense, its spiritual predecessor. Other than that, two **demos from the developers of BS Contact** (popular VRML/X3D player) containing city visualization are introduced, along with **Timewalk**, aspiring to be re-creation of the whole world in 3D. Since **Visualization of Prague Castle** is somewhat hybrid between a city visualization and visualization of a particular (rather small) area and its surroundings, some notable applications of the latter type are reviewed too. These include two presentations of cathedral interiors: first is **Saint Jean Cathedral** in France, second is generic unnamed **cathedral**. Next, we look at yet another two virtual walk applications that are visualizing various interesting places with high historical value from antic times.

The rated qualities of standalone applications are:

- Graphical quality
  - quality of models, textures, overall aesthetic feeling
- Visual effects
  - Does the application contain some extra visual effects which make the application appear as more realistic?
- Performance
  - speed of the application (smooth vs lagging)
- Tourist Guide
  - Does the application provide some form of navigation for users within the virtual world, possibly with information about surroundings?
- User Interface
  - visual quality, possible actions, practicality
- Quick to control
  - Is it possible to control the application naturally within a first few seconds?

2.3.1 Virtual Old Prague

Available at: `http://dcgi.felk.cvut.cz/cgg/vsp2`

Virtual Old Prague (VOP), a web application that allows walking through a virtual city stored in a remote database, was a student software project held on the Faculty of Mathematics and Physics of the Charles University in Prague, conducted by prof. Ing. Jiří Žára, CSc. and consisting of six students. The virtual world is implemented in VRML and presented on a HTML webpage, utilizing PHP for communication with mySQL database. The project started in November 1999 and was finished in 2001, resulting in the first version of the Virtual Old Prague [9]. In the following years, project continued to develop further on the Faculty of Electrical Engineering of the Czech Technical University in Prague, and in 2009, the second version was released. Last update to the VOP was in 2011.

User Interface is in the latest version of VOP divided into the two parts (see Figure 2.8):

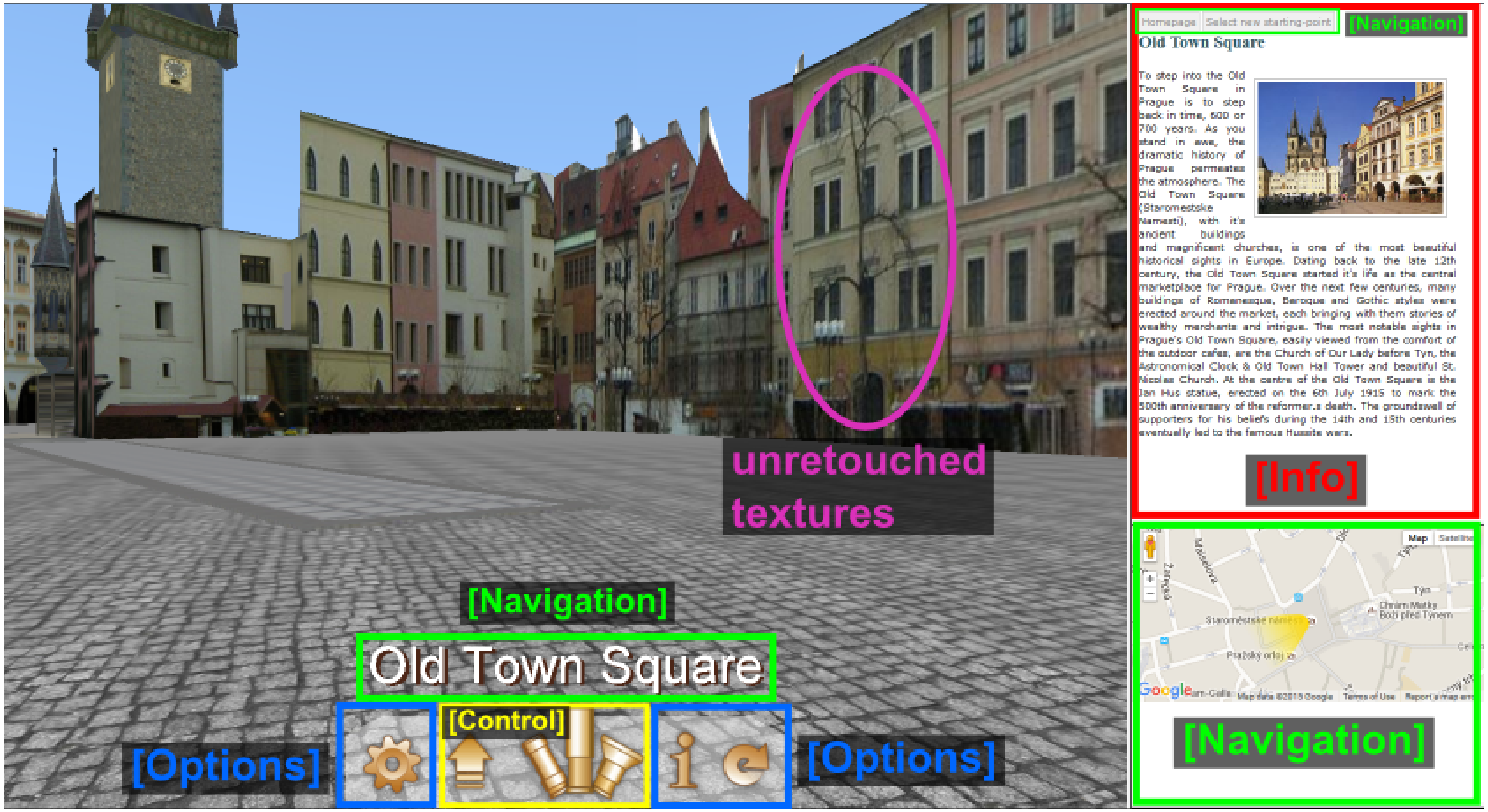
- Main frame
  - occupies a major part of the screen
  - displays contents of the virtual world through a VRML browser plugin
  - contains HUD
- Guide
  - column on the right side of the screen
  - act as a virtual tourist guide
  - provides static HTML page with information (text and images) about all objects of interest in the area
  - provides map (by Google) which shows current position of the user and his viewing direction

The tourist guide column on the right part of the screen only displays static HTML page with all (or rather some) objects of interest in the current area; it is not possible to interact with objects in the virtual world. The user can change the current location using the little navigation button which is situated above the tourist guide, which is a little misleading. It is also unfortunate that even though the viewing direction of the avatar is drawn in the map, it is not updated continuously, so it is not that useful and can be even misleading.

HUD, which is at the lower bottom part of the screen, contains five icons: the first one is for changing application options, but clicking on it closes the application and redirects user to a static HTML page (Figure 2.9), which may be very confusing. On the other hand, there is a help available for each option, which makes the process of configuration quite easy and straightforward. The options which can be changed are:

- switching between day/night
- LOD quality
- visibility range
- instant location change

A guided tour option is also available, however, it appears to be no longer working. It is possible to hide the map and the virtual guide from these options. The second button in the HUD activates the flying mode, but it is not possible to change the avatar's altitude. The third button changes camera's field of view and can be used for zooming.



Homepage | Select new starting-point

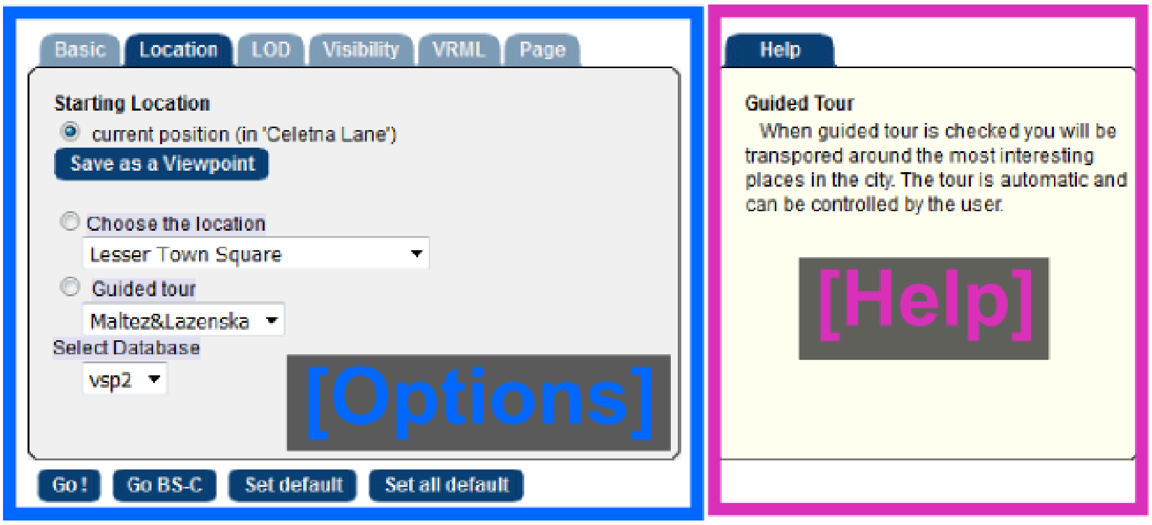
Old Town Square

To step into the Old Town Square in Prague is to step back in time, 600 or 700 years. As you stand in awe, the dramatic history of Prague permeates the atmosphere. The Old Town Square (Staromestske Namesti), with it's ancient buildings and magnificent churches, is one of the most beautiful historical sights in Europe. Dating back to the late 12th century, the Old Town Square started it's life as the central marketplace for Prague. Over the next few centuries, many buildings of Romanesque, Baroque and Gothic styles were erected around the market, each bringing with them stories of wealthy merchants and intrigue. The most notable sights in Prague's Old Town Square, easily viewed from the comfort of the outdoor cafes, are the Church of Our Lady before Tyn, the Astronomical Clock & Old Town Hall Tower and beautiful St. Nicolas Church. At the centre of the Old Town Square is the Jan Hus statue, erected on the 6th July 1915 to mark the 500th anniversary of the reformer.s death. The groundswell of supporters for his beliefs during the 14th and 15th centuries eventually led to the famous Hussite wars.

Figure 2.8. Virtual Old Prague – Old Town Square

The fourth button is for turning on the debug information intended for developers – it is questionable whether such button shall be present in the HUD, where it can be seen by regular users. The last button is for returning to the previous location, however, the technical execution is not ideal – the application is closed and then opened again (in the previous location). There are no elements in the user interface for controlling the avatar. Controls (keyboard and mouse) are the default ones of the VRML player, which is a good thing – controlling the avatar feels familiar.

The graphical impression is generally pleasing and it is obvious that the world was designed very carefully, and with performance in mind. Models are not complicated – they lack details which are often supplemented by textures (doors, windows, stairs, balconies, ...), but the important thing is that their overall geometry structure is very precise to their real world counterparts, thus leaving a good overall impression. Available locations have varying quality, though; for example the Prague Castle looks impressive, but then there are locations which exhibit downsides such as unretouched textures with trees on walls, lights in windows and similar artefacts. The resolution of textures is low, which hurts the impression if the avatar is close to objects using them. Although dated for today's graphical standards and not containing any visual effects, VOP is a great example of delivering good graphics using as little resources as possible. Performance is fine, however application can suffer from several problems related to the VRML browser which renders graphics, for example bugs in collision detection or even random crashes.



Starting Location

● current position (in 'Celetna Lane')

○ Choose the location

Lesser Town Square

○ Guided tour

Maltez&Lazenska

Select Database

vsp2

Guided Tour

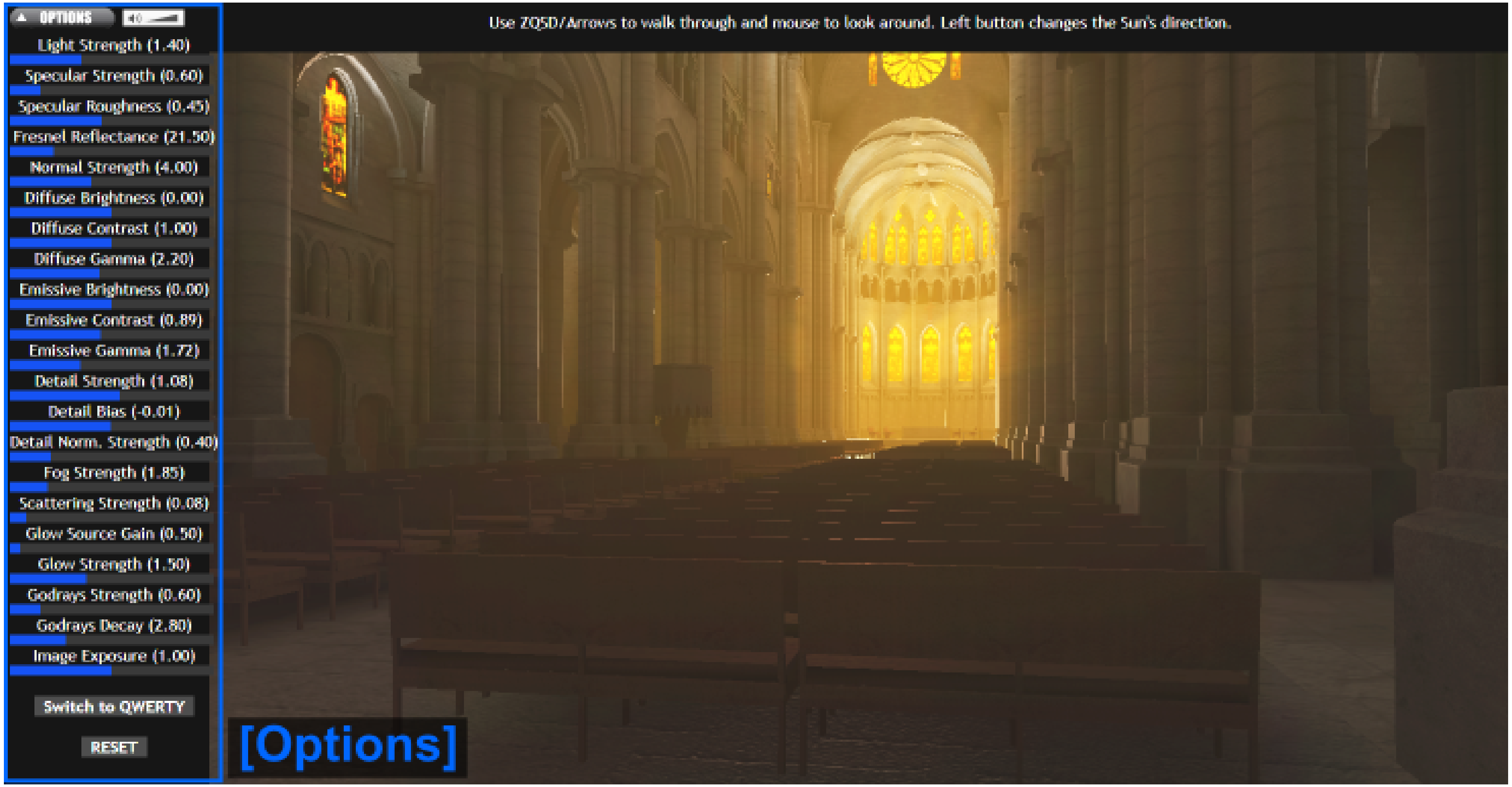
When guided tour is checked you will be transpored around the most interesting places in the city. The tour is automatic and can be controlled by the user.

Figure 2.9. Virtual Old Prague – options dialog

2.3.2 Saint Jean Cathedral

Available at: `http://patapom.com/topics/WebGL/cathedral/intro.html`

Despite the fact that the application is titled as a virtual walk, it is rather a technology demo from 2012, implemented in WebGL. It does not contain any information about the cathedral, possible objects of interest or even anything related to the actual place. Model of the cathedral is made of more than 500 000 triangles and uses many hires 2048×2048 diffuse and normal textures. Rendering features spherical harmonics radiosity, light shafts, glow and light-scattering [10]. User can change various parameters such as light and glow strength, diffuse and emissive characteristics etc. using the user interface. The application contains a collision detection, overall graphics impression is excellent and it runs smoothly.



Use ZQSD/Arrows to walk through and mouse to look around. Left button changes the Sun's direction.

Light Strength (1.40)
Specular Strength (0.60)
Specular Roughness (0.45)
Fresnel Reflectance (21.50)
Normal Strength (4.00)
Diffuse Brightness (0.00)
Diffuse Contrast (1.00)
Diffuse Gamma (2.20)
Emissive Brightness (0.00)
Emissive Contrast (0.89)
Emissive Gamma (1.72)
Detail Strength (1.08)
Detail Bias (-0.01)
Detail Norm. Strength (0.40)
Fog Strength (1.85)
Scattering Strength (0.08)
Glow Source Gain (0.50)
Glow Strength (1.50)
Godrays Strength (0.60)
Godrays Decay (2.80)
Image Exposure (1.00)

Figure 2.10. Saint Jean Cathedral WebGL application

2.3.3 Interactive demos by Bitmanagement Software

Available at: http://www.bitmanagement.com/en/cityshowcase, http://www.bitmanagement.com/en/ellwangen, http://www.bitmanagement.de/demos/cathedral/cathedral.wrl,

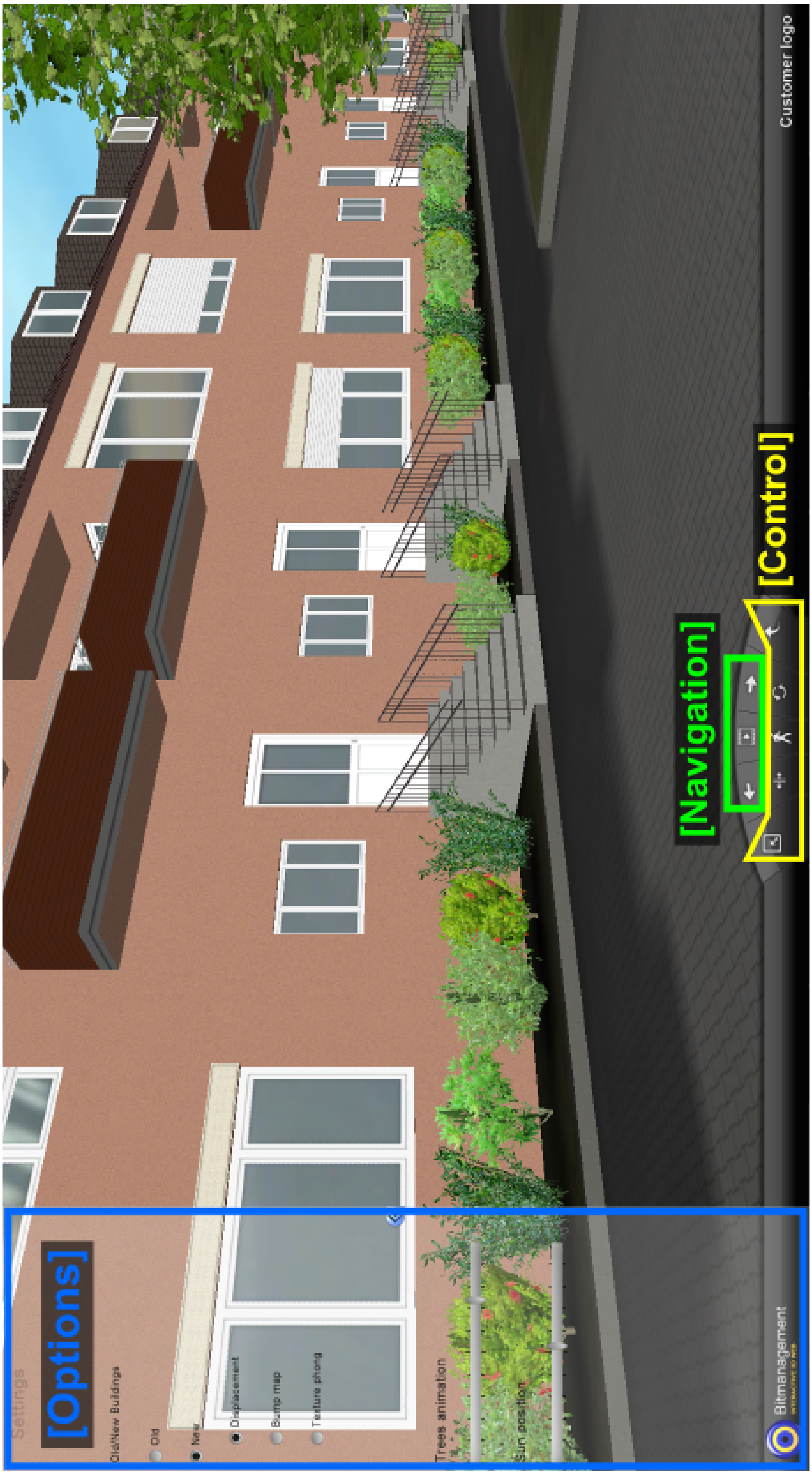
Online 3D City Visualization and **Interactive city visualization with night simulation** are two interactive demos implemented in VRML and developed by Bitmanagement Software[1]), known mainly for their cross-browser and cross-platform 3D engine BS Contact, used often for displaying VRML/X3D worlds. Graphical quality of both demos is not much visually pleasing, probably because of poor quality of textures.

In the night simulation demo, the user can adjust the settings of the application via HUD icons in the top-left part of the screen (Figure 2.12) and switch time of the simulation between day and night – the night scene looks noticeably better, but the price is high: FPS drops significantly. Otherwise, both demos run smoothly. Controls are the default ones of the VRML player, which is alright. On the other side, the user interface (which is also more or less the one of the VRML player) is not that impressive (Figure 2.11). Its visual quality is arguable. As for its layout, all buttons are in the center of the screen. When the mouse is hovered over a button, its caption is displayed. There are eight buttons in total:

- three for changing the avatar's moving modes
- two for jumping to the previous/next viewpoint
  - it is not possible to choose specific viewpoint
- one which executes kind of “fly tour“
  - the avatar is moved from a viewpoint to viewpoint in smooth transitions
  - cannot be turned off by clicking on its button, nor it can be turned off by clicking on the button for changing the avatar's moving modes, which is kind of confusing (we have to press ESC keyboard button to turn off the fly tour)
- one for “fit to screen” functionality
  - positioning avatar so that the entire virtual world is in his field of view
  - the practical use is questionable
- one for returning avatar's view to the usual position (i.e. without any head tilt)
  - the practical use is questionable

There are not buttons for navigation of the avatar in the virtual world. There are also additional options available in the left part of the screen (can be shown or hidden) mentioning sun position, tree animation and probably model quality, but none of these options seem to actually affect anything at all.

[1]) http://www.bitmanagement.com



Old/New Buildings

○ Old
● New

● Displacement
○ Bump map
○ Texture phong

Trees animation

Sun position

Figure 2.11. Online 3D City Visualization – Bitmanagement

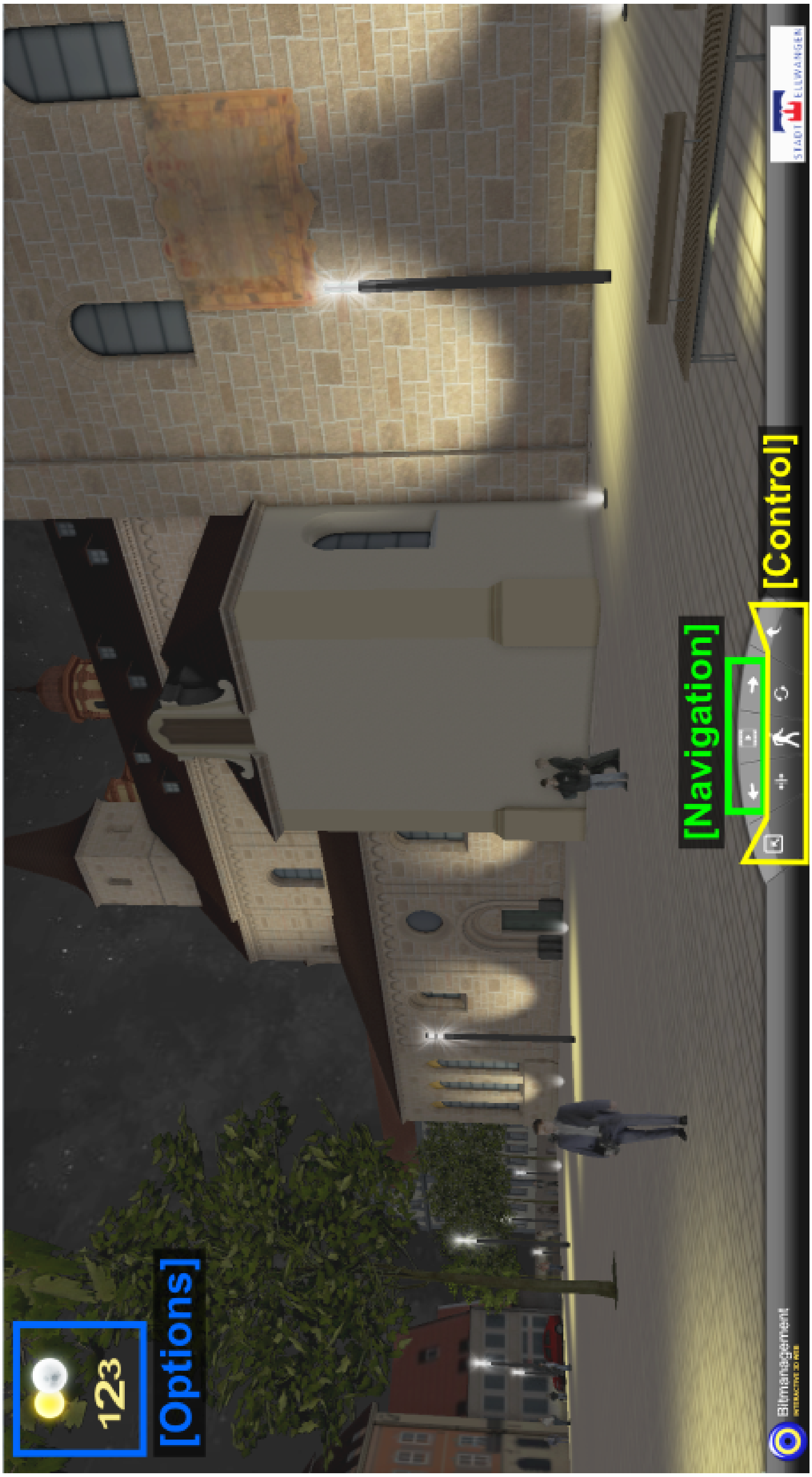


Figure 2.12. Interactive city visualization with night simulation – Bitmanagement

Third VRML demo by Bitmanagement Software entitled **Cathedral** is looking way much better than the city visualization demos. In fact, it is on par with the Saint Jean Cathedral demo, with nice light effects, quality textures and smooth performance. That being said, it also shares all downsides such as no tourist guide and no user interface. Controls are standard depending on the VRML player.

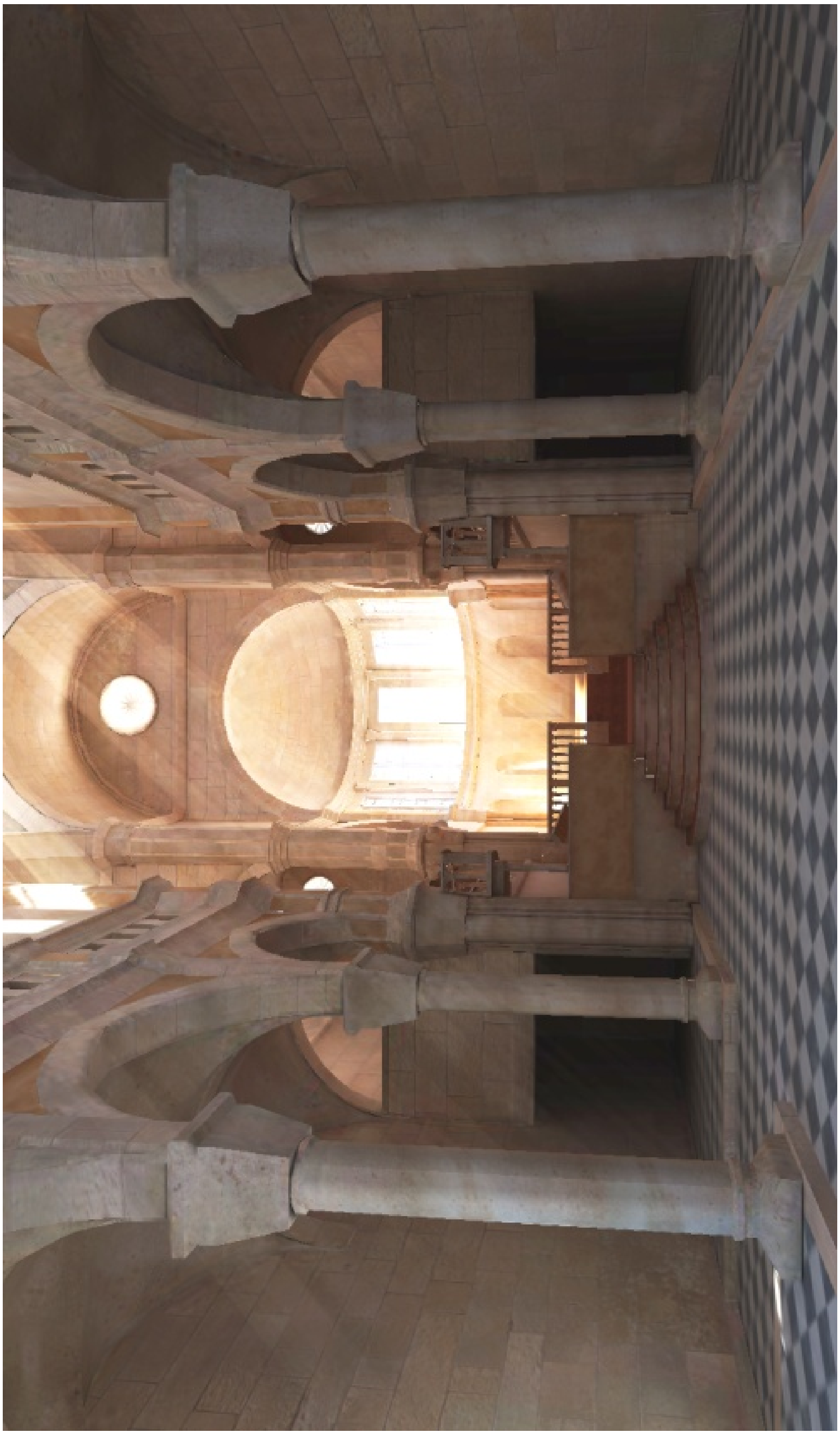


Figure 2.13. Cathedral – Bitmanagement

2.3.4 3D tours by Internet Dominion

Available at: `http://jerusalem.com/tour`, `http://vatican.com/tour`

Internet Dominion[1]), which describes itself as a corporation specializing in internet related products that support and service individuals and companies, developed series of over ten virtual 3D walks which show replicas of sites in **Jerusalem** and in **Vatican**. The tours are implemented in Flash and have good performance. They are virtual walks in their "true" sense – there is a professionally narrated voice tourist guide who talks either about neighboring objects of interest or generally area where the avatar currently is. The application has collision detection.

Some places contain an interactive 3D icon symbols (Figure 2.14) which can be clicked on and used to play a so-called *movies* (= animated viewpoints), which are again voice narrated. During a movie, the application performs a fly-over over the place when playing it, occasionally rotates the view around and performs other similar movement characteristic for animated fly-overs. Downside is an absence of a text version of the voice-narrated tourist guide; only an extremely brief information about the area is displayed in the right part of the screen. User can also display a map (bottom image on Figure 2.16), which shows current location of the avatar and allows teleportation to various places in the area. The last interesting option is called *Autopilot*, which is mode where the user is passively watching as the avatar walks automatically by itself over the whole place, visiting interesting places and triggering movies.

The user interface is nice looking and quite well arranged (Figure 2.15). The window dialogs with the help and the map can be seen on Figure 2.16. The graphical impression is very good, and even shadows are present, although static. The environment is carefully modeled with pretty textures, which creates great overall feeling about the virtual world. The controls are quite common – moving with the keyboard, looking with the mouse – and easy to pick up right away, but looking around can be confusing at first, because the mouse axes are reversed, so moving the mouse in the left-top direction actually moves the view to the right-bottom direction and it is not possible to change this behavior; it also not possible to control the avatar other way than using the keyboard or a mouse, and its moving speed is painfully slow.

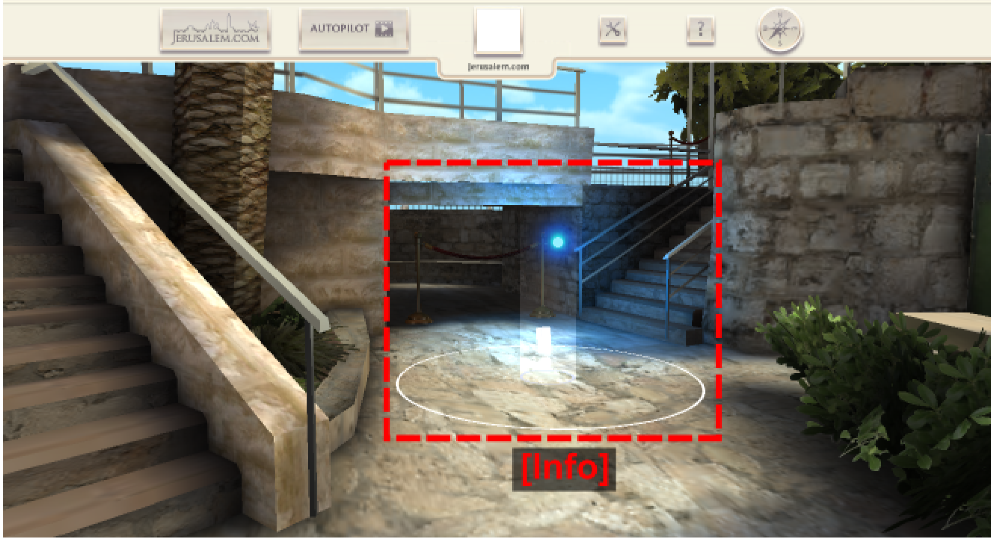
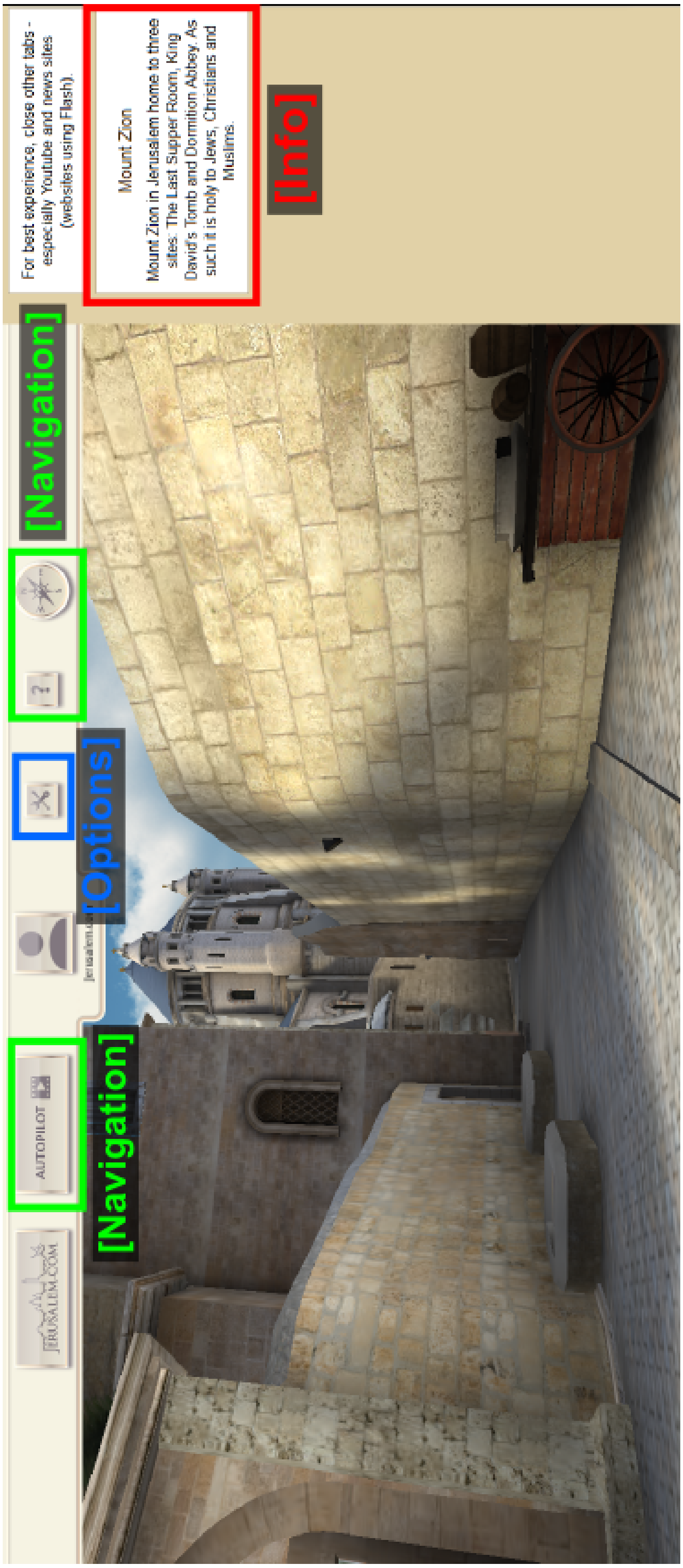


Figure 2.14. Internet Dominion – interactivity in 3D

[1]) `http://internetdominion.com`

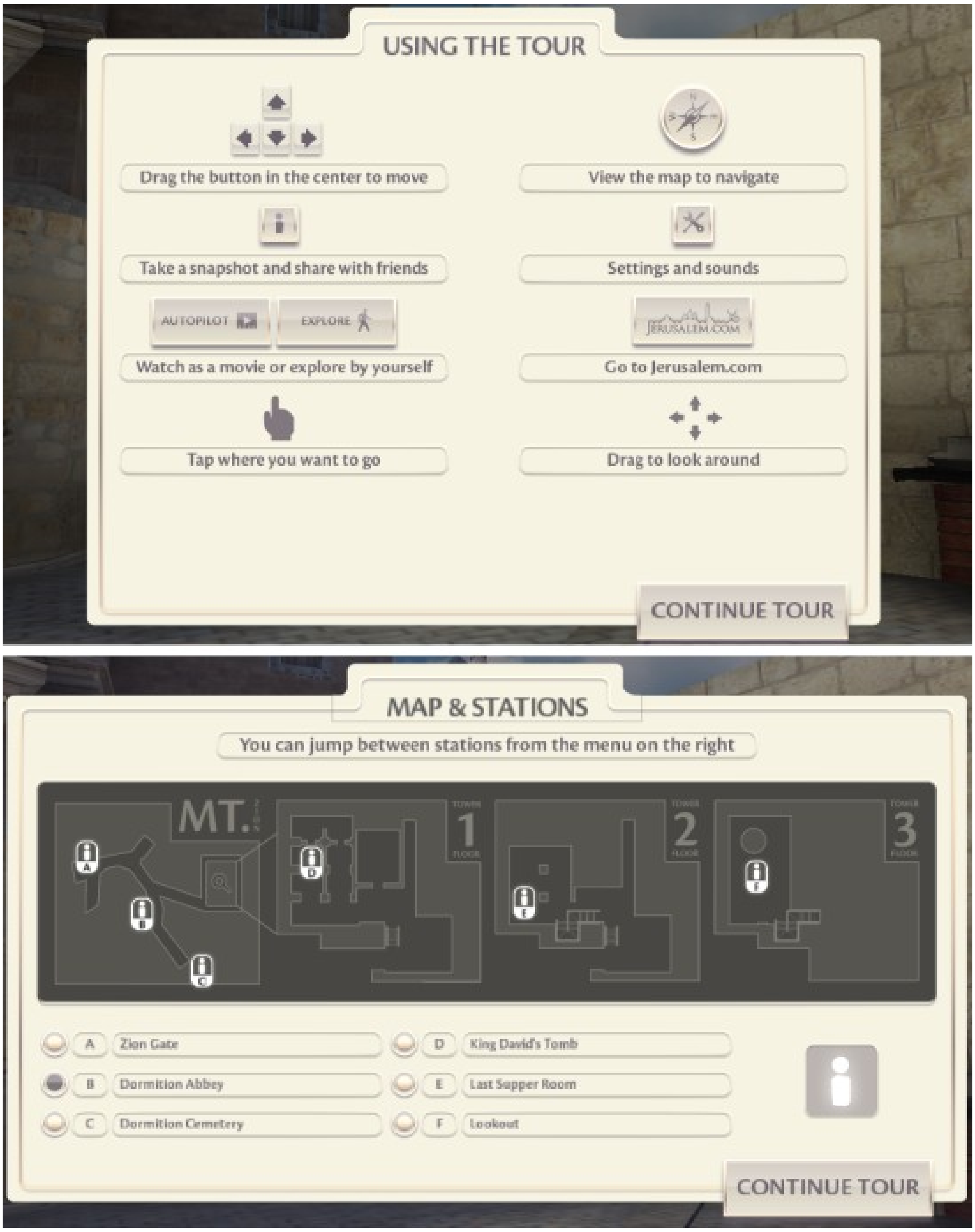


For best experience, close other tabs - especially Youtube and news sites (websites using Flash).

Mount Zion

Mount Zion in Jerusalem home to three sites: The Last Supper Room, King David's Tomb and Dormition Abbey. As such it is holy to Jews, Christians and Muslims.

Figure 2.15. Internet Dominion – Jerusalem



USING THE TOUR

Drag the button in the center to move

View the map to navigate

Take a snapshot and share with friends

Settings and sounds

AUTOPILOT EXPLORE

Watch as a movie or explore by yourself

JERUSALEM.COM

Go to Jerusalem.com

Tap where you want to go

Drag to look around

CONTINUE TOUR

MAP & STATIONS

You can jump between stations from the menu on the right

○ A Zion Gate

● B Dormition Abbey

○ C Dormition Cemetery

○ D King David's Tomb

○ E Last Supper Room

○ F Lookout

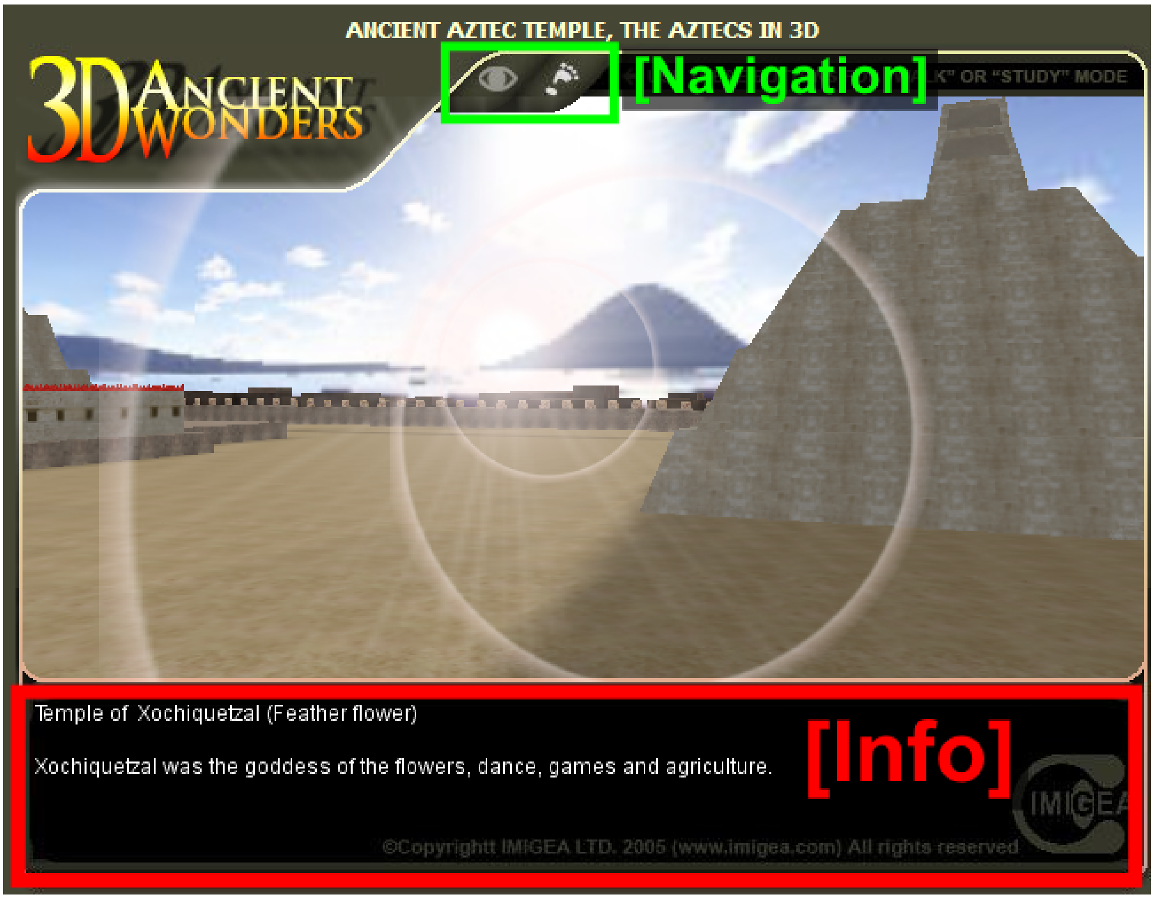
CONTINUE TOUR

Figure 2.16. Internet Dominion – UI window dialogs

2.3.5 3D Ancient Wonders

Available at: `http://www.3dancientwonders.com`

3D Ancient Wonders is a series of virtual tours dedicated to recreating ancient relics and buildings (Gaza pyramids, Stonehenge, ...) in 3D virtual reality. Developed by Imigea[1]), the electronic learning material and online game development company. Application can be seen as a sort of virtual museum, allowing the users to examine relics and buildings as a virtual model or alternatively explore them by means of a virtual walk. Sometimes, when entering a certain area or being near an object of interest, a very brief text tourist guide in the lower part of the screen appears, telling bits of information about the place. Application is from 2005 and implemented in Adobe Shockwave.



Temple of Xochiquetzal (Feather flower)

Xochiquetzal was the goddess of the flowers, dance, games and agriculture.

©Copyrightt IMIGEA LTD. 2005 (www.imigea.com) All rights reserved

Figure 2.17. 3D Ancient Wonders – Imigea

It runs smoothly, contains a collision detection and the feelings regarding its graphic quality can be described as mixed – the application's resolution is very low, some textures clearly did not survive ravages of time (e.g. skybox texture), but there are visual effects such as (static) shadows or lens flare which make a nice impression. Unfortunately, the user interface is exceptionally unpleasant, and becoming familiar with controls takes a while. The user can only switch between an examination mode and walking mode. Mouse look is not supported.

[1]) `http://www.imigea.com`

2.3.6 Timewalk

Timewalk[1] company works on the yet-to-be-released application that goes by the same name, which resembles widely known and popular online virtual world Second Life[2]. It aims to be a realistic and interconnected 3D virtual clone of every building and street on Earth, and more importantly, a social media (and business) platform. Main method for obtaining such a vast amount of models is similar to that which Google Earth used in past – contribution of the users.

Since the application has not been released yet, further information is not available, except a few trailers and short interview clips with developers available on YouTube channel[3] of the developers. The graphical quality is not very impressive and judging from the interview[4] with technical director of 3D modeling, it does not seem it will ever be, since developers try to keep a polygon count very low so the Timewalk would be able to run even on entry-level devices. Because the Timewalk is not playable at the moment, it was not rated, nor included in the comparison of all standalone applications.

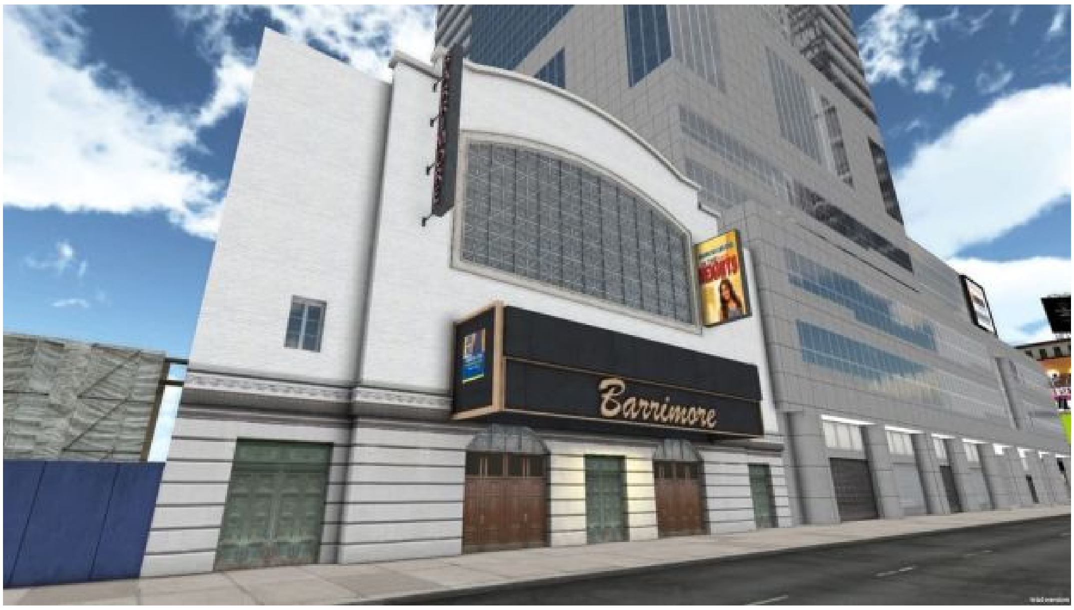


Figure 2.18. Timewalk [11]

[1] http://timewalk.tw

[2] http://secondlife.com

[3] http://www.youtube.com/channel/UC8v8On-4fBurkGTL6gAXhig

[4] http://www.youtube.com/watch?v=3AXNZ7UHrv8

2.3.7 Comparison of standalone applications

Weight	Criterion	VOP	SJ	City	Night	Cath	DOM	AW
0.25	Graphics quality	0	+	-	-	+	+	0
0.10	Visual effects	0	+	-	0	+	+	+
0.20	Performance	+	+	0	-	+	+	+
0.20	Tourist Guide	0	-	-	-	-	+	0
0.20	User Interface	0	-	-	-	-	+	-
0.05	Quick to control	+	+	+	+	+	+	-
	TOTAL SCORE	**63 %**	**60 %**	**15 %**	**10 %**	**60 %**	**100 %**	**53 %**

Application	Abbreviation
Virtual Old Prague	VOP
Saint-Jean Cathedral	SJ
Bitmanagement – City	City
Bitmanagement – Night City	Night
Bitmanagement – Cathedral	Cath
Internet Dominion	DOM
Ancient Wonders	AW

Table 2.2. Comparison of standalone visualization applications

The ratings for each application and their overall rating can be seen on Table 2.2. The first place undisputedly belongs to the **3D tours by Internet Dominion**, which excelled at every rated quality. The second place is occupied by the **Virtual Old Prague**; its dated graphics and an average user interface were detrimental for its rating. The third place is joint with **Saint Jean Cathedral** and **Bitmanagement Software Cathedral** – although looking very good, they lack an appropriate user interface and a tourist guide. **Ancient Wonders** demonstrate some advanced visual effects like shadows or lens flare, and apart from some texture quality issues and low resolution of the application, it looks nice. The user interface and controls are horrible, which is a shame, since it has at least some tourist guide, compared to the better rated Saint-Jean Cathedral or Bitmanagement Software Cathedral applications. The **city visualization demos by Bitmanagement Software** placed last in the comparison, which is no surprise, since they do not posses any impressive qualities.

2.4 Conclusion of similar systems analysis

City visualization systems are currently dominated by map services applications and focus on displaying maps in 3D, using methods of computer vision and automatic object reconstruction from aerial images. Created worlds are suitable for exploring from a distance; the user quickly obtains the idea how the place and the surrounding area looks like in the real world. However, these worlds are not suitable for virtual walks due to their lack of detail (low quality textures, geometric artefacts). The automatic object reconstruction method cannot handle some objects very well – either because of their position (they are not exposed enough), size (too small/thin) or complicated geometry (statues).

It is certainly possible to create a virtual walk with small amount of manual work. Street view-like photos could help with automatic reconstruction of finer details of objects, could be used for source of good quality textures and so on. There is quite a probability that this will be the next step in city visualizations employed by developers of map applications in the future. However, all inconveniences of the automatic reconstruction would appear again. Not to mention that the less ordinary the place is – with many small/complicated objects and possibly with parts inaccessible by machine carrying photographic equipment – the worse would be the output. For a high quality virtual walk which aims to be as much realistic as possible, it is inevitable that some manual work is performed.

In our concrete case of visualization of the Prague Castle area, even if we had means to perform an automatic reconstruction from aerial photographs or had access to such data, it would not help us that much. It would help with rough outline of models – precise distances and ratios between objects would be kept, but other than that, there would not be much benefits over classical photograph-assisted modeling, i.e. using satellite photographs for creating outline and using manually taken photographs for modeling details. Therefore, the manual work is a must in our case; we have to take photographs of the castle which can be used for modeling and we have to model it ourselves.

As for technology inspiration that can be picked up from 3D maps, an interesting observation is that each 3D map functionality that has been reviewed is implemented using WebGL, so we should definitely keep an eye on this technology (more on this in the following Chapter 3).

We can certainly pick up some inspiration from the user interfaces and available features of 3D maps. An ability to explore a city from a distance is very convenient one and would make a nice addition for a classical virtual walk application. It is important that this view is completely adjustable in this sort of “flying” mode, because as we have learned by analyzing e.g. Google Maps, it is frustrating if the choices for view customization are limited. Also, having marked objects of interest in the classical map is a very convenient feature, and we cannot forgot to mention on-screen buttons which allow controlling the view to the virtual world using just the mouse.

There are just a few **standalone presentations concerning pure city visualization**. Exception to this is the Virtual Old Prague, the spiritual predecessor of our application, which provides a priceless lesson about how a virtual walk around the area of Prague Castle can be successfully realized, but also what could have been done better. The rest of the standalone presentations revolving around city visualization are either company showcase demos or business ventures. **Presentations focusing on visualization of small area or building** are more common and some impressive visualizations such as the Saint

Jean Cathedral can be found. Nevertheless, presentations which are actually a virtual walk (in a sense of guiding the user and providing various information) are rare to come by. What is worse, a lot of these virtual walks are outdated technologically and probably would not be able to visually satisfy nowadays users. Fortunately, there are exceptions to this such as the Jerusalem and Vatican 3D tours by Internet Dominion, which provide another valuable example what qualities should good virtual walk exhibit.

Analysis of standalone presentations has shown us several things. For the graphical impression to be satisfying, the most important is having models with enough portion of details and – even more importantly – quality textures. Extra visual effects such as shadows bring the look even closer to the reality and certainly improve the overall graphics quality. A well-designed user interface and virtual guide mechanism are vital for a good user experience. An application should strive to be easily controllable right away; it is advisable that the user interface contains a HUD which provides an alternative way of controlling the application to the usual keyboard option. A clickable 2D map of the virtual area is also a nice feature which helps the user with orientation in the area. Lastly, more advanced users would certainly welcome if they can adjust settings of the application (e.g. time of the simulation – day/night, controls, etc.).

Chapter 3
Technology analysis

This chapter deals with the choice of a technology which will be used for the application implementation. A vast number of technologies available for the realization exists. Most of them leaves us with a prescribed set of programming language and way how to render graphics, but we have also an option of more custom approach using WebGL for rendering with any programming language that can be compiled into JavaScript (because WebGL has JavaScript API). We will evaluate analyzed technologies based on specified criteria and decide which will be used for the implementation.

3.1 Goals

Our goal is to analyze available technologies for implementation of a 3D virtual walk application running in a web browser and evaluate them. The technologies will be evaluated based on the following criteria:

- Necessity of a web browser plugin installation
    - Does an application implemented in a given technology require installation of an external plugin by the user?
    - no installation is the best
- Browser support
    - ideally, the majority of web browsers should be able to run the technology
- Community, Future
    - Does the technology have an active community (discussion boards, tutorials, ...)?
    - the potential of technology to be "alive" in the future (e.g. maintain at least non-decreasing amount of developers interested)
- Web integration
    - how easy/hard is to interlink the application written in the given technology with the web environment (web page, existing JavaScript libraries, etc.)
- Performance
    - performance related mainly to the 3D graphics
    - the faster, the better
- Programming
    - the similarity of programming language of the technology to the to the common languages used for developing 3D applications (e.g. to the C/C++)
- Development tools
    - the quality of IDEs available

  - overall experience during the development
- Debugging
  - difficulty of debugging
- Vendor dependence
  - Is the development and evolution of the technology itself tightly tied to one company or "anyone" can participate?

The system of rating (-, 0, +) and the total score computation is same as in Chapter 2. The ratings will be shown after all technologies are reviewed.

3.2 VRML

Virtual Reality Modeling Language is a standardized technology used for describing 3D world and aimed specifically for the web environment. The first version from 1994 allowed static worlds, and the second version from 1997 added interactive behavior and scripting (Java, JavaScript). In order to execute VRML files, the user needs a program called *VRML browser* such as **BS Contact**[1]) or **Cortona3D Viewer**[2]). It is also possible to embed VRML files into web pages and run them from a web browser if a VRML browser plugin (provided by vendors of VRML browsers) is installed. VRML worlds are easy and quick to create, thanks to the simple syntax and the fact that developer does not need to be concerned with rendering or things related to virtual walk mechanisms (walking with the avatar, collision detection) – it is done automatically by the VRML engine. It can be said that an emphasis is on a design rather than programming and even a developer not skilled in computer graphics is able to create rich virtual worlds. Unfortunately, this "design approach" has negative performance consequences, developer does not have a control over the rendering process and integration of an advanced application logic can be quite clumsy. On top of that, the users need to have a browser plugin installed to see the VRML content and VRML browsers exhibit buggy behavior, use a lot of computer resources and crash. Also, the ability to use them in the future is uncertain, as web browser vendors have been gradually removing support for external browser plugins [12–13].

3.3 X3D (X3DOM)

Extensible 3D is an open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML, developed by X3D Working Group. It is a successor to the VRML [14] and an ISO ratified standard (first version in 2003) that provides a system for the storage, retrieval and playback of real time graphics content embedded in applications. X3D aims to incorporate the latest advances in graphics hardware features and architectural improvements and is fully compatible with VRML 2.0. Scenes are usually encoded using XML, but syntax of VRML 2.0 or binary encoding can be used as well. X3D supports multi-pass/multi-stage texture mapping, pixel and vertex shaders, deferred rendering and many more graphical effects.

Embedding and execution of X3D files works same as in case of VRML files. Luckily, developers of X3D came up with a concept allowing to run X3D application in browsers

[1]) http://www.bitmanagement.com/products/interactive-3d-clients/bs-contact

[2]) http://www.cortona3d.com/cortona3dviewer

without any plugin called **X3DOM**, where graphics are rendered by WebGL. X3DOM is an open-source framework and runtime, a way how to integrate X3D into a webpage. It includes X3D elements as a part of HTML DOM tree, allowing developer to manipulate a 3D content just by adding, removing, or changing DOM elements, which brings many nice features (e.g. support of `onclick` event on 3D objects) [15]. Sadly, not counting academic and research use, X3DOM has not found its way to the users, just like VRML. Apart from browsing the web and looking at number of search results (either in general or at discussion servers like Stack Overflow[1])), we can see its low popularity by looking at the number of followers at GitHub[2]), where source code of X3DOM and other open source technologies is available. X3DOM has around 300 followers, while a technology for similar purpose called Three.js (will be mentioned later) has around 22 000[3]).

3.4 Adobe Flash

Adobe Flash is a platform for creating wide range of applications and content that run across multiple operation systems and devices, with origins in 1996. Applications are programmed in ActionScript, an enhanced superset of ECMAScript. A **Flash Player** (runtime for browsers) needs to be installed on a device for an application to run. More than 98% of internet users have Flash Player installed, however from November 2011, Adobe has discontinued supporting it on mobile devices [16]. As for desktop computers and laptops, although web browsers vendors have been removing support for external plugins, they made an exception in case of Flash. Flash applications would still be able to run – at least for now. When a Flash application is ready to deploy, it is compiled into one SWF file which can be run in Flash Player.

3D hardware accelerated rendering can be achieved using Stage3D API, which is more abstract than native APIs (e.g. OpenGL, DirectX) and thus more distant from hardware itself [17]. Shaders written in Stage3D are intended to be rather simple.

It is also important to note that there is a strong vendor dependence on Adobe with Flash, since there is no Flash player with source code publicly available and with a license that permits reuse. There are projects like Gnash[4]) or LightSpark[5]) which try to tackle the task, but they do not support all features and their updates are mostly from over two years ago.

3.5 Google Native Client

Google Native Client, developed by Google since 2010, is an open-source technology for running native compiled code in the browser, with the goal of maintaining portability and safety. Developer can write application targeting multiple operating systems (Windows, Linux, Mac, Chrome OS) and CPU architectures (x86, ARM), with code running at near-native speed [18]. Applications can utilize threads, use low-level rendering API (OpenGL ES 2.0) and other things similar to classic desktop application. A Native Client web application consists of JavaScript, HTML, CSS, and a Native Client module written in a language supported by the SDK provided by Google. Currently supported languages are C/C++.

[1]) `http://stackoverflow.com/`

[2]) `https://github.com`

[3]) Statistics regarding followers can be seen on `https://github.com/search?q=stars:%3E1&s=stars&type=Repositories`

[4]) `http://gnashdev.org`

[5]) `http://lightspark.github.io`

For compilation of C/C++ code into a module loadable by browser, SDK provides either GCC or LLVM[1]). The latter one, which is recommended and being used more frequently, produces one portable module. At the runtime, browser uses ahead-of-time translator to translate produced module into a native code for architecture of the user's machine. Currently, the only browser capable of running Google Native Client applications is Google Chrome, but at least no plugin installation is required. From the statements of Mozilla's vice president of products Jay Sullivan [19] and Opera's chief technology officer Håkon Wium Lie [20], it is unlikely that other browsers will ever support Google Native Client applications.

3.6 WebGL + JavaScript compilable language

A choice which provides the most freedom. WebGL is currently the only standardized way how to render 3D on web pages, and JavaScript is the only client-side web scripting language supported by virtually all browsers (unlike, for example, VBScript that is supported only by Internet Explorer). It is natural that WebGL has JavaScript API, thus combination of WebGL + JavaScript for running a 3D application is the most straightforward option. But the development does not have to happen in JavaScript, thanks to using a special type of compiler called *transcompiler* (alternatively *source-to-source compiler*). Transcompiler is a type of compiler that takes a source code written in one programming language as its input and outputs the very same source code, but written in another programming language [21].

This brings many new possibilities and advantages such as reusing already written code, developing in familiar programming language and even an ability to debug application like a classic desktop one in the favorite IDE and so on. There are some catches, though – language features from source (input) programming language which are not available in JavaScript (output) must be emulated, which can hinder performance. Needless to say, the integration with existing JavaScript libraries and APIs becomes a significantly more difficult task. There are mechanisms allowing a developer to call JavaScript from source programming language and vice versa, but getting it to work and with a minimum performance overhead is not always easy or even possible. Another disadvantage could be fine tuning performance-critical applications (such as 3D applications). Developer has to look at the code which actually runs rather than the source code and since transcompiler produces JavaScript code which looks a lot different than the source code, it can be a grueling task to find a performance bottleneck. It is fair to mention that tools for JavaScript performance tuning and debugging have come a long way and are still improving, but they are yet to reach the maturity of tools available for classical programming languages, so the less time spent in debugger and profiler, the better.

A few relevant choices of source programming languages (frameworks) which support compilation into JavaScript are discussed. These are **TypeScript** (an extension of JavaScript), **Dart** (a whole new scripting language), **Google Web Toolkit** (Java) and **Emscripten** (C/C++).

3.6.1 TypeScript

TypeScript[2]) is a programming language developed by Microsoft in 2012 and it is a superset of JavaScript, adding static typing and classes. Any program valid in JavaScript

[1]) `http://llvm.org`

[2]) `http://www.typescriptlang.org`

is valid in TypeScript as well. It is also possible to use external JavaScript library through *definition files* (resembling header files from C/C++), where types are assigned to variables and such. Then, all library objects can be treated like regular, typed TypeScript objects [22]. The definition file making is quite an ordeal (it must be done manually), but fortunately definition files for many popular JavaScript libraries are already done and publicly available[1]).

3.6.2 Dart

Dart[2]) is an open source web programming language developed by Google, first appeared in 2011, and can be seen as a JavaScript rival. It is an ECMA standard[3]) since 2014. Dart is an object-oriented, supports classes, single inheritance and static typing. WebGL can be used through internal Dart library.

It can be transcompiled into JavaScript or run native in Dart Virtual Machine at full speed. However, only Chromium browser[4]) (development version of Chrome) has Dart VM included. Benchmarks from [23] show that execution in Dart VM is roughly $1,5\times$ faster than original JavaScript code, and transcompiled code runs usually about just a bit slower. JavaScript can be called within Dart through special object `JsObject` – a name of the desired function or variable is passed into its constructor and then `JsObject` acts as a proxy to desired JavaScript object.

According to the [24], Dart VM will not be integrated into Chrome, ever. Brendan Eich's (former Mozilla CEO) statement [25] implying that neither other browser vendors plan to integrate Dart VM then comes as no surprise.

3.6.3 Google Web Toolkit

Google Web Toolkit[5]) is a set of tools that allows creation of front-end applications in Java, including the user interface. Originally developed by Google with initial release in 2006, now fully open sourced project maintained by the GWT Steering Committee. The communication with external JavaScript libraries (or execution of pure JavaScript code) is handled similarly as in TypeScript. In order to use some external function, developer has to create a *"native"* function, which acts as a wrapper. Return type of a native function and type of its parameters must be specified. The body of a native function is then written in JavaScript (and can also call external JavaScript code). The usage of WebGL is possible through libraries which wrap JavaScript API for WebGL.

3.6.4 Emscripten

Emscripten is a transcompiler which compiles LLVM code to JavaScript, first version released in 2011 and developed by Alon Zakai from Mozilla. Generated code is highly-optimizable JavaScript in **asm.js** format [26].

Asm.js[6]) is a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. The asm.js language provides an abstraction similar to the C/C++ virtual machine: a large binary heap with efficient loads and stores, integer and floating-point arithmetic, first-order function definitions, and function pointers [27]. It provides a major performance boost for web applications which are written in a

[1]) `http://github.com/borisyankov/DefinitelyTyped`

[2]) `https://www.dartlang.org`

[3]) `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-408.pdf`

[4]) `https://www.chromium.org/Home`

[5]) `http://www.gwtproject.org`

[6]) `http://asmjs.org`

statically typed language such as C/C++, by limiting language features to those which can be optimized in the process of ahead-of-time compilation.

With Emscripten and asm.js, it is possible to port C/C++ applications (even containing OpenGL – calls are translated into WebGL equivalents) to the web and run them at speed approaching their native performance – benchmark from [28] shows that web version code is just $1,5\times$ slower. Moreover, since asm.js is nothing more than JavaScript, it is already supported on all browsers (unlike e.g. Google Native Client). Using Emscripten, some of the popular C/C++ games and engines were already ported to web, such as Unreal Engine 4 [29].

There are several ways of interaction with JavaScript from Emscripten C/C++ applications. Most straightforward (and with an overhead) is calling special function `emscripten_run_script()` with a JavaScript code as an argument. More efficient, yet still simple is to declare desired JavaScript function in the C/C++ code with an `extern` keyword.

3.7 Comparison of technologies

In this section, we shall analyze the rankings and decide which technologies will be used for our implementation. The ranking of evaluated criteria for each technology and their overall rating can be seen on Figure 3.1. The ranking can differ in the "WebGL + JS compilable language" case based on which language is chosen. This is reflected by "/" between the marks.

Weight	Criterion	VRML	X3DOM	Flash	NaCL	WebGL&JS
0.15	Plugin	-	+	0	+	+
0.18	Browser support	+	+	+	-	+
0.13	Community, Future	-	0	0	-	+
0.10	Web integration	-	+	0	0	+/0
0.18	Performance	-	0	0	+	+
0.08	Programming	-	0	0	+	0/+
0.08	Development tools	+	0	+	+	-/+
0.05	Debugging	+	-	+	-	-
0.05	Vendor dependence	+	+	-	-	+
	TOTAL SCORE	**36 %**	**72 %**	**63 %**	**54 %**	**83-90 %**

Table 3.1. Comparison of technologies

Considering the goal of a maximum performance, **VRML** is not an optimal choice, plus there is the problem with an external plugin support in web browsers. **X3D** in the form of **X3DOM** solved many shortcomings of VRML and runs in a browser without external plugin. But unfortunately, it is not geared towards the goal of maximum performance, it is intended for applications that just want to utilize some 3D functionality, which was proclaimed even by its authors in X3DOM documentation [30]. Also, there is not much of a community around it, which is another thing which keeps rating of X3DOM from being higher. There is no objective advantage of using **Adobe Flash** over other technologies, plus there is a vendor dependence on Adobe, since no up-to-date public open source Flash player exists. And even though Flash web browser plugin got an exception from browser vendors for now and is allowed to run (compared to other external plugins), there is no guarantee for the future. **Google Native Client** with its near native speed and possibility to use features common for desktop applications is surely an interesting technology and more than a suitable candidate, however, the only browser supporting it is Google Chrome and it will probably stay this way forever, which puts this technology out of question.

The combination of **WebGL** and language which is capable of compilation into **JavaScript** provides the highest performance and compatibility, which are both crucial aspects. It is no surprise this option placed first in the overall ranking. But now the question is: *Which JavaScript-compilable programming language will be used with WebGL?*

3.7.1 Choosing JavaScript-compilable programming language

In this section, we will come to the decision about which JavaScript-compilable language will be the most suitable for use with WebGL. The ranking of evaluated criteria for each language and their overall rating can be seen on Figure 3.2.

Weight	Criterion	JS	TS	Dart	GWT	Emscripten
0.10	Programming	-	0	0	+	+
0.15	Development Tools	0	0	0	+	+
0.05	Debugging	0	0	0	+	+
0.25	Performance	0	0	0	0	+
0.20	Possible problems	+	0	0	-	-
0.10	Web integration	+	0	0	-	-
0.15	Community, Future	+	0	-	-	-
	TOTAL SCORE	**68 %**	**50 %**	**43 %**	**43 %**	**55 %**

Table 3.2. Comparison of JavaScript-compilable languages

Dart has no advantages over the other choices – it is similar case as Google Native Client. It does not run natively even in Chrome, only in Chromium. **Google Web Toolkit** with Java would bring similar development advantages as using Emscripten with C/C++, but the performance of generated JavaScript would not be so fast, making it inferior choice. **Emscripten** has proven competent in the field of performance-critical applications and the development experience would be hands down the best one with the ability to use classic programming language and mature IDE like Visual Studio, but there are other things to consider such as scale of the application. Our application is not meant to be the next cutting-edge 3D game engine, so there is a chance that developing in C/C++ and using Emscripten could bring more troubles than advantages, e.g. weird behavior or bugs can occur in transcompiled JavaScript, which can be very hard to track and even harder to fix.

Based on the evaluated criteria and the analysis above, the **language chosen for implementation is pure JavaScript**. This choice provides easy interaction with external JavaScript libraries and APIs, and full control over the application code, which will be useful for searching speed bottlenecks during the performance tuning. The last technology question to answer is: *What middleware (framework) above WebGL will be used?*

3.7.2 Middleware above WebGL

It has been decided that the application will run under WebGL + JavaSript, but there is one more choice left to be made – whether to work with WebGL directly or use some kind of middleware above it. It is possible to use one of many Javascript frameworks which simplify interaction with WebGL, i.e. abstract away its low-level nature. Three popular frameworks are briefly introduced, and along with raw WebGL, the decision between these four choices are made, based on the following criteria:

- Performance
  - the ability to achieve fastest performance possible
- Flexibility
  - the ability to customize the system to a specific need
- Prepared solutions
  - possibility of using parts of code (algorithms, mechanisms, classes, data structures) which can be easily plugged into the application and used
- Community, Documentation, Examples

Lets now briefly introduce frameworks appearing in the comparison: **Three.js**[1]) is a WebGL framework for general use, probably the most widespread one, very popular (20-30th top interesting repository on GitHub with approx. 22 000 followers). **Scene.js**[2]) is another popular framework, aimed toward doing high-performance 3D visualization, and specialized towards fast rendering of large numbers of individually articulated objects [31]. **Babylon.js**[3]) is similar to Three.js, although more geared towards games.

Weight	Criterion	Raw	Three.js	Scene.js	Babylon.js
0.25	Performance	+	0	0	0
0.15	Flexibility	+	0	0	0
0.40	Prepared solutions	-	+	+	+
0.20	Documentation, Examples, Community	+	+	0	0
	TOTAL SCORE	**60 %**	**80 %**	**70 %**	**70 %**

Table 3.3. Comparison of middleware above WebGL

The ranking of evaluated criteria for each framework and their overall rating can be seen on Figure 3.3. Naturally, frameworks cannot reach such level of performance and flexibility as raw WebGL tailored to specific application needs, so it can come as a surprise that raw WebGL has placed last, but the reason is simple: it does not provide any prepared solutions whatsoever, which is the most important rated quality. We would have to do absolutely everything by ourselves, which is extremely time-demanding when developing an application of scale which our application will have. Scene.js and Babylon.js share the same place and both have one disadvantage compared to Three.js, which has placed first – they do not have such strong community, thus also working with them would be harder.

[1]) `https://github.com/mrdoob/three.js`
[2]) `http://scenejs.org`
[3]) `http://www.babylonjs.com`

3.8 Conclusion of technology analysis

Various technologies available for the implementation of our virtual walk application were discussed and evaluated based on the defined criteria (community and browser support, performance, tools available, ...). **WebGL** was chosen as the technology for displaying 3D graphics, mainly because it is a web standard and provides a good performance, and **JavaScript** was chosen as the language in which will be the application written, because apart from the fact that it is prototype-based language (not your average language in which 3D applications are written) and have "only" acceptable quality of available tools, it is the lingua franca of the client-side web programming, making integration with the web in general seamless. Similarly, several options of middleware above WebGL were analyzed and evaluated as well. Because the emphasis in evaluation was put on the feature of prepared solutions (which can greatly reduce development time), **Three.js** was chosen as the middleware above WebGL.

Chapter 4
Project specification

The task of our project is to present the Prague Castle as a 3D model on the web in a form of virtual walk, with focus on (balance between) the performance and graphical quality. This means we have to create new models or possibly reuse already existing ones from the Virtual Old Prague project[1]) and develop a web application which is able to efficiently display these models and provide virtual walk functionality. In this chapter, we will summarize the chosen technology approach and specify the requirements for the application.

4.1 Choice of technology

Based on the analysis from the previous chapter, we have chosen **JavaScript + WebGL + Three.js** as the technologies which will be used for the development of the application (along with HTML + CSS, which is given, since our application is web-based). Using JavaScript instead of languages which are compiled into it reduces the potential sources of bugs introduced by an added layer of abstraction and thus makes for easier process of debugging and performance tuning as well. Hardware accelerated rendering inside the browser is possible without any additional external plugins thanks to the fact that WebGL is a standardized web technology; this is very important from both technical and user standpoint, because browser vendors have been gradually dropping support for external plugins ([12], [13]) and most of the users do not like to be bothered by a plugin installation anyway. Using Three.js as the middleware above WebGL allows us to save incredible amount of time by providing proven rendering solution which is still flexible and can be mostly bent, one way or another, to suit our needs.

4.2 Application requirements

According to [32], the requirements are divided into the two distinct categories – Functional (describe what the application should do) and Non-functional (characteristics and potential constraints of the application). The main source of the requirements is the knowledge obtained in similar visualization systems analysis (Chapter 2) and technology analysis (Chapter 3).

4.2.1 Functional requirements

Apart from the already collected ones, requirements have been also enriched by the recommendations from [33]. The functional requirements for our application follows; please note that reasoning behind the inclusion of some listed requirements will be explained later (mainly in Chapter 7 which discusses the User Interface).

[1]) Since the VOP project was held on the FEE CVUT and supervised by prof. Ing. Jiří Žára, CSc. as well, we have been kindly granted an access to the data of the project

- Virtual walk functionality
  - application shall allow user to virtually visit the Prague Castle located in Prague, Czech Republic
  - application shall provide users with an additional textual information about interesting objects in a form of text articles, possibly containing images or hyperlinks
- Navigation
  - avatar shall stop before obstacles and collide with the environment (e.g. walls, terrain which is too steep)
  - user shall be able to navigate the avatar in the two modes:
    - **WALK**: shall resemble walking like in the real world, the avatar falls down when there is no ground below
    - **FLY**: shall allow flying (like a bird)
  - avatar should be able to tilt its head at custom direction (in other words, the application should support mouselook)
  - application shall contain 2D map synchronized with the user's movement in 3D
  - application shall allow direct choosing of target places (viewpoints) from a list or a map
- User Interface and Input
  - user shall be able to control the application with the keyboard or mouse
  - application shall contain graphical user interface with buttons for each possible action
  - application shall provide help subsystem (hints, advices, ...)
  - it should be possible to change the application's settings (e.g. mouse sensitivity)
  - application shall be available in Czech and English language localization
- Graphics quality
  - application shall employ modern visual effects (shadows[1]), ...) and contain new, improved 3D content (new models and high-resolution textures)

4.2.2 Non-functional requirements

- The application shall run in majority of web browsers[2]) (Mozilla Firefox, Google Chrome, Opera) and use technologies provided by HTML5 specification, without need for installing any plugin
- The development shall be carried out in JavaScript, with Three.js framework as WebGL middleware
- The application shall achieve playable framerates (30+ frames per second) on desktop PCs manufactured in 2009+, and on laptops manufactured in 2011+
  - [35] claims that the average age of PC around the world is between 5 to 6 years, so we consider this required hardware specification as fairly reasonable
- The application should look similar on all resolutions which are likely to be present at the user machine [3])

[1]) Affected by the time of the day

[2]) Firefox, Chrome and Opera covered 87,9 % of the user base in October 2015 [34], which we consider as enough

[3]) That being said, we do not expect any user machine to have a resolution lower than 480×320 – correct appearance on smaller resolutions is not guaranteed

Chapter 5
Software structure

We begin this chapter with the discussion of software architecture requirements for our virtual walk application and touch upon the subject of applied design principles. The resulting structure is explained, providing brief explanations for concrete components (classes).

5.1 Architecture requirements

Software architecture in general is focused on organizing components to support specific functionality. The nature of our application is very different from applications which fall into the categories of enterprise software, office suite software etc. and is more similar to the category of games/game engines. Claiming that we can turn a virtual walk into a 3D action game if we add some enemies and ability to shoot them is an oversimplification, and our virtual walk application is certainly not as complex as a game or an engine, but it nicely proves the point that they share a lot, which is why we have decided to follow a component-based architecture employed in these kind of applications. We have based our components on game engine architecture outlined in [36] (visual representation can be seen in Appendix on Figure B.1 and B.2) and kept only components relevant to our application. The visual representation of our application's component-based architecture can be seen on Figure 5.1, where components drawn with blue background represent purely external libraries (not authored by us), components with green background represent partly external code (some parts authored by us) and finally components with white background represent purely internal code (code completely authored by us).

In order to minimize costs and possibly maintenance requirements, and to promote usability and extensibility, we stick to key principles of software design [37], mainly:

- Separation of concerns
  - dividing application into distinct features with as little overlap in functionality as possible
  - minimization of interaction points to achieve high cohesion and low coupling
- Single Responsibility principle
  - each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality
- Do not repeat yourself
  - we should only need to specify intent in one place, i.e. specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component

Essentially, our goal is to make the design flexible so that when a change is made, only the relevant portion of a code is replaced. Object-oriented programming paradigm is

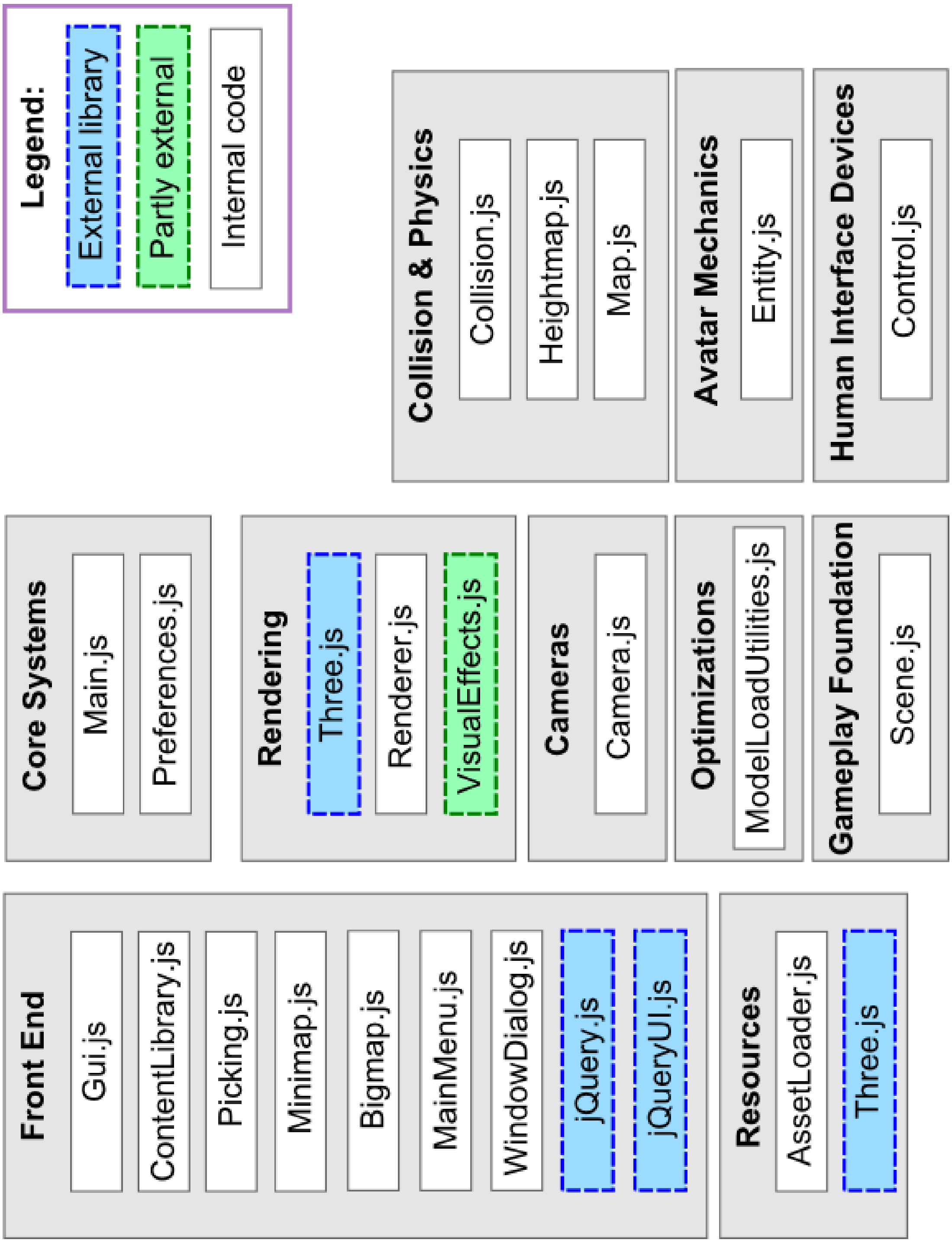


Figure 5.1. Architecture of the application

used heavily. We should also mention that any modifications of used libraries (especially Three.js) should be avoided if possible; potential upgrade to a newer version of library is then deadly simple, comparing to a situation where custom added code has to be merged with a newer version.

5.2 Components

The classes of the application are divided into the several components based on their purpose:

- **Core Systems**
  - `Main.js`
    - application entry point
    - downloads all necessary assets and initializes the whole application
    - implements main loop (function repeating each frame, ensuring drawing and virtual world updates)
  - `Preferences.js`
    - holds various constants and setting options (e.g. shadows on/off, path to asset directories, language, ...)
- **Rendering**
  - `Three.js`
    - external framework used for WebGL rendering
    - the reason why we do not have any `Mesh`/`Model`, `Shader` classes etc. – they are already present in `Three.js` and we can extend them if necessary
  - `Renderer.js`
    - contains various methods associated with drawing on the screen
    - most of methods are utility methods (reading color of the pixel from an image, making a snapshot of the screen, ...)
    - the actual rendering of the virtual world is delegated to the `Three.js` renderer
  - `VisualEffects.js`
    - implements visual effects (shadows, ambient occlusion, ...)
      - the code (including shaders) of individual visual effects is mainly external (more on this in Chapter 10), with occasional modifications tailored for our application
    - specifies the rendering process (i.e. configures rendering passes depending on which effects are turned on/off)
- **Optimizations**
  - `ModelLoadUtilities.js`
    - optimizes models upon loading into the application for the best performance
    - Chapter 9 dedicated to the performance optimizations explains in a further detail
- **Front End**
  - `Gui.js`
    - main class responsible for initialization and manipulation with the user interface

  - `ContentLibrary.js`
    - manages all strings (including their translations) appearing in the application
    - explained in a greater detail in Chapter 7
  - `Picking.js`
    - implements the technique which allows finding out a 3D model which is currently under the mouse
    - more on this in Chapter 7
  - `Minimap.js`
    - functionality related to the small (classical, paper-like) map that is permanently visible on the screen
  - `Bigmap.js`
    - functionality related to the big map which can be opened from an on-screen HUD
  - `MainMenu.js`
    - implementation of the application's main menu
  - `WindowDialog.js`
    - implementation of a general window UI dialog
    - concrete dialog windows extend it with their specific needs
  - `jQuery.js`, `jQueryUI.js`
    - external libraries used for building the user interface

- **Avatar Mechanics**
  - `Entity.js`
    - used for representing the user's avatar
    - contains reference to a `Camera.js` instance (how the avatar sees the world), current mode (walking or flying) and methods for switching between, several properties related to collision detection etc.

- **Cameras**
  - `Camera.js`
    - implements our application-specific camera behavior (mouse look)
    - connected with the `Three.js` camera which takes care of standard camera computations (computing projection and view matrices etc.)

- **Human Interface Devices**
  - `Control.js`
    - handles entire input processing from the user (keyboard, mouse, ...)

- **Collision & Physics**
  - `Collision.js`

    - specifies the interface for resolving collisions between the avatar and the virtual world
    - will be discussed in Chapter 8
  - `Heightmap.js`
    - system used for collision detection implementing the interface of `Collision.js`
    - will be discussed in Chapter 8
  - `Map.js`
    - computations related to the projection of the avatar's position from the virtual 3D world to a 2D image and vice versa
    - used by `Heightmap.js` and also by the `Minimap.js` and `Bigmap.js`
- **Resources**
  - `Three.js`
    - used for loading the scene and models (`ColladaLoader.js`)
  - `AssetLoader.js`
    - used mainly for sophisticated downloading of files from the server
    - we can give it a task of downloading certain file, it allows us to specify dependencies between the tasks (implemented as a directed acyclic graph and utilizing topological ordering) and callbacks which shall be executed upon various events (file has been downloaded, all task dependencies are finished, ...)
    - a task can be anything (e.g. execution of some function), it does not have to be some file downloading, although that is the usual case
    - because our application is a web application and data transfer is asynchronous, the dependency functionality is absolutely vital; for example, we use this to make sure that the application does not start before the data necessary for collision detection to work are downloaded

Chapter 6
Virtual content creation

Virtual content – both 3D (models) and 2D (textures) – for a virtual world must be somehow created. As for 3D content, although methods for an automatic reconstruction from photographs (photogrammetry) have been gaining popularity in recent years, using such methods for creating virtual content suitable for virtual walks is problematic – a good reconstruction needs a lot of suitable photographs and a quality software which performs the actual reconstruction. There are some free solutions, but they do not produce such results as the commercial ones. In any case, a further manual processing is usually needed, because the reconstruction produces unoptimized meshes with a lot of polygons, some parts missing etc. Because of all these difficulties, it is no surprise that modeling in a 3D modeling program is the gold standard for creating a content suitable for interactive applications such as virtual walks and we will use this method as well.

In the same way, personally taking photographs and editing them in an image editing program is the gold standard for creating textures. This chapter will discuss available choices for 3D modeling and image editing programs and present the reader with our approach to virtual content creation in a form of a guide aimed at developers possessing a beginner's level of experience in these fields. The guide describes how to recreate a real-world objects as fully textured 3D models using a popular 3D modeling program SketchUp 2014 and an image editing program GIMP 2.8, and explains a reasoning behind choosing these programs. How to recreate an environment around a virtual area (i.e. terrain) using available elevation data from the real world will be described as well. Other addressed topics include importing the created 3D content into the application by choosing a suitable 3D format and investigation concerning possible reuse of the content from the Virtual Old Prague project. In the very end of the chapter, we publish and comment on time and file size statistics for the content creation of our application.

6.1 Goals

Our ultimate goal is to describe the process of capturing the real world place (in our case, the Prague Castle) and recreating it virtually. We shall provide a guide covering the most important steps (personal visit to the place and photographing, drawing inspirations from other sources than personal visit, creating textures, modeling the place) and focus on possible pitfalls. The guide shall be drawn on our experience during creation of the content for the Visualization of Prague Castle, so it is not to be aimed at experienced modelers (this also applies to creation of the 2D content – image editing), but rather at those with a beginner's level of skill, i.e. typically developers who are focused on the programming. We hope to provide those with a content-creation knowledge that will help them with decisions regarding the programming part of a 3D application (e.g. decisions about optimizations).

First, we shall decide on programs that will be used for creating the 3D and 2D virtual content. The desired qualities for a **3D modeling program that will be used for creating the 3D content** are:

- Active community
  - a lot of tutorials, discussion forums, an ability to find the solution to a problem etc.
- Fast to learn
- Easy manipulation
- Export to a non-proprietary format (e.g. OBJ, Collada, ...)

While we are at the topic of choosing a 3D modeling program, lets also talk about another important thing: **choosing a 3D format which will be used for importing the created virtual content into the application**. We have decided on the following requirements:

- Non-proprietary format
- Fast and simple to load by the application
- As little additional processing of the 3D file(s) as possible
  - ideally, we shall be able to export the whole scene from a 3D modeling program into one file that can be imported right away into the application

On the subject of 2D content creation, we shall choose **a suitable image editing program**. We need for it to have:

- Active community
  - a lot of tutorials, discussion forums, an ability to find the solution to a problem etc.
- Standard advanced features
  - layer support
  - perspective correction
  - filters
  - plugins
  - retouching tools
  - color balance tools
  - ability to draw arbitrary thick lines or circles
  - etc...

Another thing which falls into the category of the 2D content creation are **textures**. From support and performance reasons, we shall decide on the following requirements:

- Resolution of a texture (maximum resolution, width and height restrictions)
- Approximate total graphics card memory occupied by all application textures

Lets also not forget that since the Prague Castle is not built in the middle of nowhere, but it rises over the city of Prague, we shall account for this and **recreate the environment around the castle** at least to some degree and describe how to do that.

We shall also define the **extent of our virtual content creation** and investigate a possibility of reusing the data from the Virtual Old Prague project.

6.2 Choosing the content creation programs

Given our requirements for a 3D modeling program and the target audience of our guide, the most suitable program, without any doubts, is the **SketchUp**[1])[2]). Previously owned by Google and now by Trimble Navigation, a mapping and navigation equipment company, it is aimed for a wide range of drawing applications (architectural, civil/mechanical engineering, interior design). While other popular programs such as Blender, 3ds Max etc. are certainly more powerful and have a strong community of users, their learning curve is very steep and since they are aimed at general modeling, manipulation with tools for the purpose of creating buildings (which is what we are after) is not the most straightforward thing. On the other side, SketchUp is exactly aimed at the buildings creation (so the manipulation with tools is fast), has a large community as well and above all, is fast to learn and allows exporting to various formats, including the non-proprietary ones. It is available in both free and commercial version; the free version misses some advanced functionality, but fortunately no functions needed by us. Furthermore, some missing functionality can be even supplemented by using various available plugins. Choosing SketchUp has also another advantage: **the whole scene can be exported from SketchUp into Collada format (DAE), which can be directly loaded by Three.js into our application**. This meets all the requirements set for choosing the 3D format used for importing the created virtual content into the application. While it is true that Collada is generally not suited as a runtime format but rather as an exchange format (in contrast to, for example, glTF[3])), it is completely sufficient for our application.

Choosing an **image editing program** suitable for the 2D content creation is a simple matter. The most notable programs in this field are Adobe Photoshop and GIMP, which both comply with the requirements we have set. With both being equal for our purposes, we have chosen to go for **GIMP**[4]) (version 2.8), because we have more previous experience with it.

6.3 Extent of the content creation

We have intended to reuse (at least some) data from the Virtual Old Prague project, but upon a further investigation, we have concluded that it would be more efficient to create brand-new models and textures. Virtual Old Prague data is exclusively in VRML format. Importing VRML files directly into SketchUp is not supported and despite the fact that various converters from VRML to other formats exist – e.g. **meshconv** [5]), they fail on most of the Virtual Old Prague VRML files. So automation of the process is not possible. Although we theoretically could convert all the VRML files manually (or write our own automation tool), we have decided that such ordeal would not be worth the effort – the textures are most of the time unusable[6]) due to their low resolution anyway and models do not contain any complex one (apart from the really well-done model of the St. Vitus' Cathedral) which would be hard to reproduce; moreover, since

[1]) `http://www.sketchup.com`

[2]) We have chosen the most recent version at the time of writing – which was version 2014 – but the previous/next versions are expected to work for our purpose as well.

[3]) `https://www.khronos.org/gltf`

[4]) `https://www.gimp.org`

[5]) `http://www.cs.princeton.edu/~min/meshconv`

[6]) With a few notable exceptions which we made use of, for example sprite of the gate in The First Courtyard

models for a single object in Virtual Old Prague are often scattered through many files and contain not negligible amount of duplicated textures under different names, even further manual work would have to be performed for those models to work at least somehow efficiently in our application. Under these circumstances, we have decided it is better to create a whole new virtual Prague Castle instead. The disadvantage of this approach lies in the fact that recreating the whole Prague Castle is not possible within a time dotation allocated for the diploma thesis, so we have decided to limit the extent of the content creation to the following areas of the Prague Castle:

- Part of the Hradčany Square in front of the First Courtyard
- The First Courtyard
- The Second Courtyard
- The Fourth Courtyard
- The Garden on the Bastion

6.4 References for the content creation

Before we begin with the content creation, we have to gather some reference material of the modeled area. In this day and age, we have the luxury of having a variety of information sources. These include:

- Personal visit
- Images of the area found at the internet in general
- Online maps
- Panoramic images (street view)
- Aerial and bird eye images
- 3D model of the area

The availability of the listed sources differs depending on the area. In our case of the Prague Castle, all listed sources are available. This is extremely handy and we shall leverage this advantage during the creation of the area's 3D representation. Anyway, nothing beats a personal visit of the area, which should be the cornerstone of all content creation. We recommend repeated visits (number depends on the size of the area) in order to be familiarized with the place and to have the own imaginary 3D model of the place in mind, because it is hard to get right the exact placement, distance ratios, overall feeling etc. only from photographs or even 3D models (usually created using photogrammetry) – they contain artefacts and generally do not display smaller objects (e.g. fountains) in a way which would be of any use.

6.5 Textures creation

6.5.1 Hardware limitations

Quality of textures is vital to a pleasing visual experience from the application. A good-looking texture must have sufficient resolution. The larger a texture can be, the better, but we have to keep hardware limitations in mind:

- Texture resolution

    - Maximum supported resolution

    - we need to keep textures under the largest resolution which can still be displayed by most of the machines
    - we have decided for our limit to be 4096×4096px[1]), which is supported on 99,4 % machines according to [38]
  - Width and height restrictions
    - textures which are not "*power-of-two*" are not recommended from performance reasons (no mip-mapping, not as efficient storage in the memory, ...)[39]
    - thus we make sure that all textures have both width and height sizes equal to some power of two

- Graphics card memory
  - unlike when stored on the HDD, textures take generally a lot more space when in the memory of the graphics card
    - e.g. one 1024×256 texture may take just 60KB in the JPEG format, but 1024KB in the graphics card memory (uncompressed RGBA8 format)
  - we want to limit the amount of graphics memory used by our application
    - prevents performance drops (swapping, ...)
  - we will try to keep the total application requirements under 128-256MB for textures
    - is reasonable even for older machines

6.5.2 Acquiring photographs for textures

We will focus on creating textures from our own photographs, which is a superior method of textures creation – we do not have to worry about legal issues (license of photographs) and we can take as many photographs as we want, and have them in high-quality. Sadly, creating textures from photographs is a tedious and time-consuming work. We shall stress the importance of taking photographs which require the least amount of editing afterwards. It is better to spend a few more minutes trying to catch the "perfect" photograph angle or even postpone taking the photograph to some other time if the conditions are not in our favor (weather, too much people in front of the object, etc.).

With that in mind, it is wise to think over the time of our arrival on the place where the photographing will take place. Places like the Prague Castle are generally full of tourists during the day. While this is not a problem when taking photographs of some objects – for example statues, which are usually placed at higher places, it is a serious problem when we need to photograph objects which can be easily occluded, e.g. building facades. We recommend to arrive on these kind of places either late (generally 2 to 3 hours before the sunset) or early (after the sunrise) to avoid large groups of people.

The photographing process itself is quite straightforward:

- Setup the camera
  - either manually set ISO, exposition and other parameters

[1]) From now on, we will omit the px unit when talking about texture/image resolution

  - or leave it to the automatic mode (not optimal, but acceptable most of the times and faster)
- Position the camera
  - ideally, we want photographs of objects from the front view, i.e. an imaginary line going from the camera is perpendicular to the plane where the object's front is
- Wait for good conditions
  - ensure that there are as little as possible people and animals in the view
- Take photographs
  - better to take more photographs to counter accidental bad focus of the camera etc.

Apart from taking photographs for the purpose of creating textures, it is also practical to take photographs of the place for purposes of modeling reference. These are useful later during the modeling of the place in a 3D software because they help with recalling how the place looks (proximity of objects, overall appearance and feeling). Unlike photographs for textures, these can have people in them, have bad lighting and so on, since we will not be doing anything with them but looking. An useful practice is to cleverly place some reference object in the view (e.g. a bottle) so we can eventually roughly estimate distances/measurements from the photographs later – invaluable when modeling objects like fountains or pillars, which sizes cannot be obtained from any other sources (e.g. maps), because they are too small in comparison to the whole area.

6.5.3 Creating textures from photographs

Creation process of a texture has several steps:

- Correcting the image (photograph) characteristics
  - perspective correction (always needed)
  - lens distortion correction (sometimes needed)
  - color correction (sometimes needed)
- Selecting an area of the image as the source for the texture
  - the result of this step will actually be the first (rough) version of the texture
    - in the following steps, we will work only on this area
  - try to select an area with a `width:height` ratio equal or close to a power of two
    - prevents distortions of the texture during the final resize and export
    - e.g. 2048×1024, but **also** 2448×1224 etc.
- Retouching
  - removing disturbing objects like wires, water drops, humans, animals, ...
  - editing parts of the image which would not look good in the texture
    - e.g. reflections on/in windows
- Making the texture tileable *(optional)*
  - almost all textures in the application are drawn as tiles

        - necessary to edit the texture so that the transitions between the tiles are not visible (will be discussed later in 6.5.5)

- Exporting
    - preserve the original texture image
        - the texture image obtained so far, usually in a format used by image editing program (e.g. `*.XCF` for GIMP), will not be modified during the export
    - the texture is scaled down to a reasonable resolution
        - make an effort to find the best price-performance ratio between the texture's visual quality and the resolution size
            - use either *Lanczos3* or *Cubic* interpolation for scaling down
            - in case of PNG export, use no interpolation
        - building facades textures usually need bigger resolutions (e.g. 1024×256), material textures look often good even in smaller resolutions (256×128)
        - make sure that the scaled-down width and height size of the texture is a power of two
    - choose the export format and finish the export
        - PNG for textures with transparency, otherwise JPEG
        - keep an eye on the exported texture file size so it does not go overboard
            - but generally, it is better to have a slightly bigger file with better image quality
            - the file size itself is not an issue in these days and average connection speeds
            - a reasonable options are PNG quality around 7, JPEG quality above 90
    - Testing
        - use the texture in a 3D modeling program (e.g. SketchUp) and see how it looks
    - Texture improving
        - if the texture needs further improvements, make them until not satisfied with the result

6.5.4 Retouching textures in GIMP

The indispensable tool in our retouching arsenal is the `Heal tool`, which can remove somewhat smaller disturbing objects from images such as wires, cracks, water drops etc., and with some experience and patience, it can be used even to remove people or bigger objects (e.g. benches).

Sometimes, the `Heal tool` is not enough, or the "ill" area is just too big to heal – in that case, we can use the `Rectangle tool` and copy&paste a neighbor area which is similar to the ill area. This causes visible transitions at edges between the pasted area and rest of the image (apart from perhaps different level of the pasted area's brightness, which shall be adjusted immediately). To fix this and remove these transitions, we can use the `Smudge tool` which blurs selected area, effectively removing sharp transitions.

6.5.5 Tileable textures

When discussing textures, we have to mention building facades. The appearance of building facades are vital to a pleasing visual experience from the application. And the most important part in making this happen is to create nice-looking textures for facades. We have two options there – use either a single texture for a whole facade or use a *tileable texture*, which is a texture which multiple copies can be placed repeatedly right after each other, borders between the copies being indistinguishable, making the copies look like a single texture rather than a collection of textures.

Using a tileable texture saves precious graphic card memory, since we can use a small texture to cover a large real world area. A simple observation reveals that most building facades are rather monotonous, repeating same elements again and again (i.e. all windows look almost the same), with just a few differences, as can be seen on Figure 6.1, which is an unedited, real-world photograph of the Second Courtyard's south side (located in the area of the Prague Castle). This makes building facades ideal candidates for usage of **tileable textures**. However, using a single texture for a whole facade has one advantage over a tileable texture: preserving even the smallest-details and differences, for example when there is a flaked plaster near one window but not near another. But the problem is that with large facades, a single texture of such facade has to be large too.

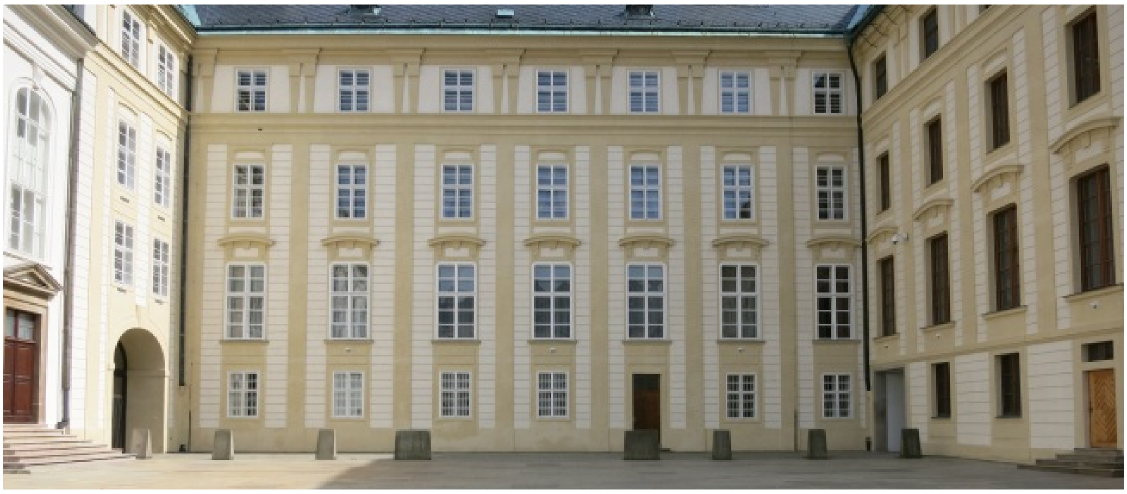


Figure 6.1. Example of a building facade in the Prague Castle (the Second Courtyard – south side)

Lets compare both approaches on the example of texturing south side of the Second Courtyard of the Prague Castle (real-world photograph was on Figure 6.1). This facade is about 25 m long, with height about 17 m. The comparison can be seen on Figure 6.1.

We suppose RGBA8 format of textures (32b per 1px). Mip-maps or any form of a texture compression was not taken into the account. *Real world coverage* means how big area in the real world is represented by the texture – it is obviously full size of the facade for the single textures, and in case of the tileable texture, it is full size of the facade as well, but after the tiling (repeating) is taken into the account. *Centimeters to pixel* ratio tells us information about the texture quality (the smaller, the better) – it shows how many centimeters from the real world are depicted by one pixel in the image[1]). To illustrate differences between different texture qualities, the first entry in

[1]) The ratios in the table have been rounded

Texture type	Tiling	Resolution (px)	Real world coverage (m)	cm/1px ratio	GPU memory (KB)
Single (VOP)	no	256×128	26,0×17,0	10,2×13,2	128
Single (our application)	no	2048×1024	26,0×17,0	1,3×1,7	8192
Tileable (our application)	7,03× horizontally	256×1024	3,7×17,0 × (7,03×1)	1,4×1,7	1024

Table 6.1. Tileable vs single textures comparison

the comparison contains a single texture with resolution which was used in the Virtual Old Prague project for this very same texture of the south side of the Second Courtyard (the texture in its original resolution is depicted on Figure 6.2).

Facades with texture quality similar to Figure 6.2 (1st table entry – single texture, VOP) look acceptable when the avatar is far away from a facade (i.e. more than 60 m), which is a problem, because usually there are facades near the avatar all the time and they are often no longer than 25 m away. Figure 6.3 (2nd table entry, single texture – our application) shows how a single texture look like in our application (although it is depicted a lot smaller, otherwise it would not fit on the page) – if a facade has texture quality similar to this texture, the avatar can stand even right in front of the facade and the visual quality is still acceptable. Finally, the last column in the comparison shows us that by using the tileable texture rather than the single one, the memory requirements are 8× smaller (with the same level of quality). The tileable texture is depicted also on Figure 6.3 – its image boundaries are colored in red, and we can see that by repeating the texture roughly 7×, we get the same image as in the case of the single texture[1]).

If we also consider the fact that for example the west side of the Second Courtyard of the Prague Castle is around 100 m long (height is still 17 m), which would mean over-the-limit 8096×1024 resolution in the case of a single texture (although we can split the facade into more parts to overcome this), **we have settled for the tileable texture approach**. The small individual differences occurring on a facade which cannot be expressed using tileable texture are small price to pay, and besides, if there is a feature on a facade which really stands out (e.g. the door in Figure 6.2), it can be added separately for example in a form of a textured quad which lies almost in the same plane as the facade or modeled as 3D element.

Figure 6.2. Virtual Old Prague texture (original resolution)

Lets now talk a little about **creating tileable textures**. Following types of tileable textures exist:

[1]) Note that the actual single texture of this concrete facade would have a slighly better quality – this is because the tileable texture repeated 7× yields a 1792×1024 texture, but 1792 is not a power of two; the single texture would need to be 2048px wide

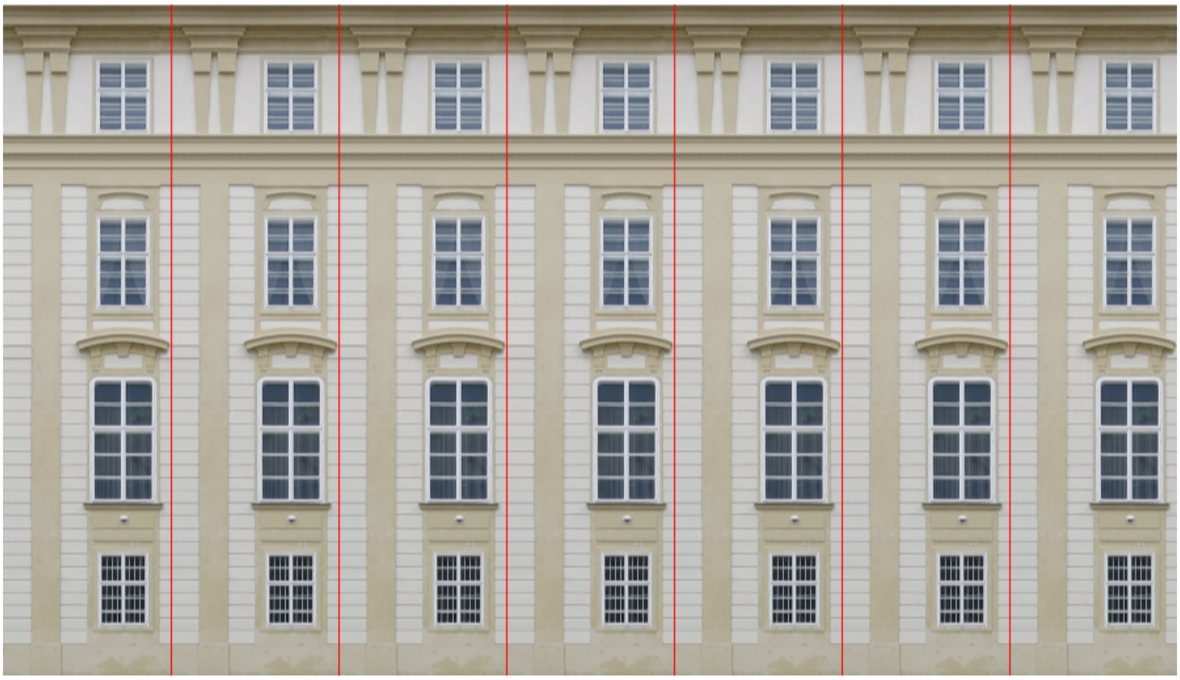


Figure 6.3. Our application texture (38 % of the original resolution, i.e. 2,6× smaller), red line delimeters texture tiles

- Horizontally tileable
  - textures can be placed side-by-side (e.g. texture tiles on Figure 6.3)
- Vertically tileable
  - textures can be placed above or below each other
- Horizontally and vertically tileable
  - textures can be placed both side-by-side and above or below each other
  - appears in the application often – all natural and synthetic materials, ground cover (plaster, grass, pavement, ...)

There are no cases of pure vertically tileable textures in the application, but there are a lot of horizontally tileable textures in a form of building facades. When creating a horizontally tiling texture for a building facade, it is a good practice to try not having a lot of parallel lines (e.g. spaces between bricks) near edges of the image. The problem is that in spite of all the effort given into the perspective correction, the image is almost always a little bit distorted (the texture on Figure 6.3 is a rare exception), which is why we have to make adjustments so that lines near both of its edges are in the same position in order to have a visually pleasing repeat effect. If possible, it is better to avoid this situation and pick a facade part which has a continuous area (e.g. plain, undecorated part of the facade) near the edges instead. The last resort solution is to copy the area near the left/right edge of the image, flip it horizontally and insert to the other edge of the image. This produces sometimes less noticeable, sometimes obviously noticeable seam between the newly inserted, flipped area and the original neighborhood of the other edge. Fortunately, the seam can be made more or less disguised and very hard to notice (if at all), for example by using blurring tools in an image program (e.g. `Smudge Tool` in GIMP as discussed in 6.5.4).

Horizontally and vertically tileable textures appear in the application very often in a form of various materials such as marble, plaster, grass, sandstone, but also as roof tiles and many more. Making tileable texture for a material is a lot easier than for facades – because materials do not have such distinguishable features and are rather monolithic, we can generate nice-looking tileable texture automatically in an image editing program. In GIMP, we can use built-in function `Make Seamless`. However, the function is rather simple and does not allow for any customization, so we get kind of a hit-or-miss result. It is better to use advanced version of this function which goes under the name `Make Seamless Advanced` and can be found in the GIMP Plugin Registry [40]. The advanced version allows specifying both horizontal and vertical overlap and a few more useful properties (gradient contrast etc.), so it is possible to experiment with these until the result is satisfying.

6.5.6 Approximating complex objects by textures

When an object has a complex geometry (e.g. a statue), it can be advantageous to approximate its 3D model by placing a texture with transparent background on a quad instead (a.k.a. sprite). Sprites are used in the application in case of statues and similar objects, because it would be very time-consuming to recreate them properly as 3D models.

The creation of a sprite does not differ much from creating a standard texture. The only extra step is to separate the target object from the background in the image and luckily, modern image editors offer tools which make this a breeze in most of the cases. In GIMP, we activate the `Magic Wand tool` (also called the `Fuzzy Select Tool`), which is capable of an image segmentation. There are several options which can be adjusted, but usually, it is sufficient to simply adjust the threshold option until the `Magic Wand`’s contours are around the object. The unfortunate cases when the magic of the `Magic Wand` does not work include images where the target object and its background visual separation is not clear enough – such image can be seen on Figure 6.4. In this case, the color of the object and the background is very similar, which causes problems for the `Magic Wand` algorithm. We are left with no choice than to **separate the object manually**, e.g. by using the `Free Select tool` and drawing the contours around the object ourselves. After the contours are drawn, we use the `Magic Wand tool` which will take care of making the object selected (this is necessary for the next step).

When the object and background are separated, it is good to paint the background in a few different colors to see how the object’s edges look. We often discover that they are lightly tinted, which would not look good in the application (Figure 6.5). To fix this in GIMP, with the object selected, we execute `Select - Shrink` and shrink the object a little (value of 2px is usually a good choice), which will remove tinted edges (Figure 6.6). The sprite is now ready for scaling down and exporting into PNG. Unlike with JPEGs, we use no interpolation when scaling down PNGs, because that would decrease the visual impression by introducing artefacts (“teared” pixels) here and there, mainly near the edges (see Figure 6.7, which is a part of the image from Figures 6.5 and 6.6). Nevertheless, even with no filtering, we should always carefully check the scaled-down image for tears – sometimes one or few pixels are looking off and should be fixed. With this, the sprite is finished and ready to use. One last thing to consider is that if the object in the sprite touches or is very near the image edges, it can happen that when the sprite is placed on a quad, we can see artifacts at the quad edges, which is nothing other than the sprite texture being repeated. To fix this, we add a blank space around the edges in the image (4-8px) and fine-tune the texture placement in SketchUp.

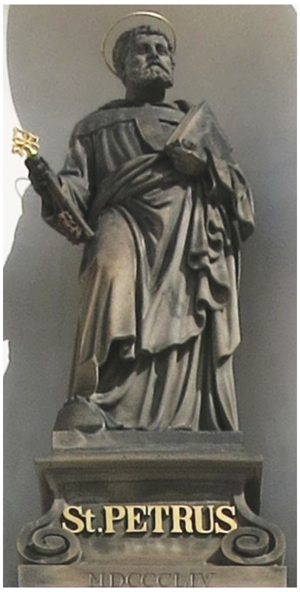


Figure 6.4. Problematic image for the automatic background separation

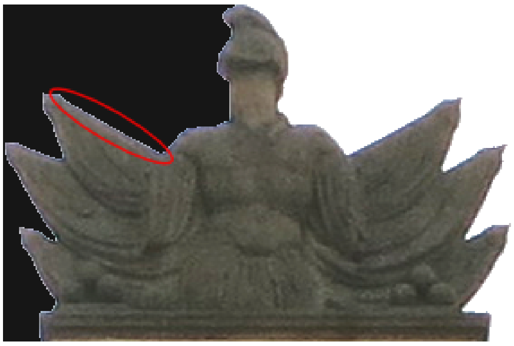


Figure 6.5. White tinted edges after the background separation (an example highlighted in red), shown on different backgrounds

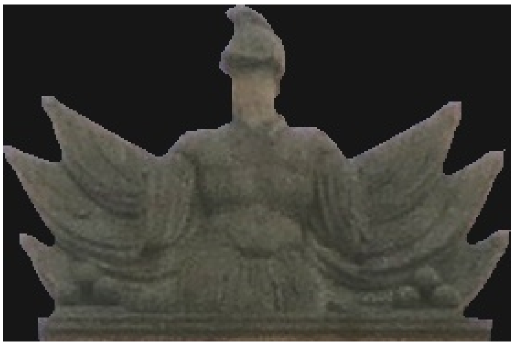


Figure 6.6. Tinted edges removed by shrinking

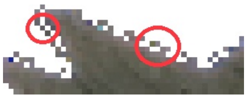


Figure 6.7. Artefacts occurring when scaling down PNG with an interpolation filter

6.6 Models creation

6.6.1 Modeling process outline

During a creation of a virtual world, we recommend sticking to the following outline during the process of modeling:

- Create a rough geometry of an entire object/area
    - do not create details or any complex geometry, because the geometry will likely change (at least a little)
        - create just the main geometry (facades etc.) and the "approximating" geometry instead of the complex one (just mark the place where the complex geometry will be)
- Progressively work on parts of an object/area
    - when the first rough version of the object's texture is created, we will immediately UV-map it to see if the geometry (or the texture itself) needs to change and perform such changes if necessary
    - if no changes need to be made or when all changes are done, the whole texture is created and mapped to the geometry and we proceed with the next part
- Create details
    - when the main geometry is guaranteed not to change anymore, we can start to work on details (i.e. adding plasticity) and create complex geometries (e.g. balustrades, passageways, ...)

6.6.2 Creating details

Complex geometries such as balustrades, passageways, stairs, fountains and similar objects are modeled exactly, using appropriate tools. We start by drawing a complex geometry or part of it from the top view, most often by using the `Line` and the `2 Point Arc tool` in SketchUp, and then use `Push/Pull tool` to extrude the created "ground plan" and make it 3D.

Adding **plasticity** refers to making surfaces which are relatively flat (e.g. facades) look 3D. When a facade has only a texture applied, it does not look quite convincing, especially when looking at it from the side. The `Push/Pull tool` comes to rescue in this case. When adding plasticity, for example to a facade, we will trace (again, using mostly the `Line` and `2 Point Arc tool`) shape of windows, their ledges, decorative arcs and other protruding elements. When the tracing is completed, the `Push/Pull tool` is used to extrude these elements and plasticity is achieved. Note that it is often good to make elements extruded a little more than they do in the reality so the effect is more noticeable in the virtual world. The good thing about adding plasticity is that we can use the original facade texture for texturing of extruded elements, just the UV coordinates need to be changed a little. This is great because the number of overall textures is not increased, since as we will learn in Chapter 9, the performance of our application depends mainly on the number of textures rather than the number of triangles.

6.6.3 Handling two-sided faces

Sometimes, faces of an object are visible from both inside and outside. Rendering faces from both sides poses a performance penalty and we want to avoid that. Thus, when a face should be two-sided, we use the `Push/Pull tool` to extrude it into the direction of its back face by some small amount (but not too small in order to prevent a potential z-fighting, 3-4cm works good) and create a second face, which will be used as the second side (the second face is rendered as one-sided as well). A great care must be taken during the extruding process, since SketchUp 2014 tends to maliciously flip the front face direction of the original face after its extrusion.

6.6.4 Productivity tips

Creating a virtual world is a time consuming process, so it is for the best if the author adopts an effective working style which can save significant amount of time and maybe more importantly, author's level of concentration. We give following brief recommendation (tips are intended for SketchUp, but they are applicable in general as well):

- General settings (SketchUp)
  - Switch to the larger tool buttons
  - For purpose of modeling a virtual world, the *Architectural Design template* works best
  - If the cursor snapping is too aggressive, it can be adjusted or completely turned off
- Keyboard shortcuts
  - absolutely vital for an effective modeling process
- Hide entities obstructing the view
  - useful when we want to work e.g. on the "inside" part of buildings
    - SketchUp zooms in/out with the center at the mouse cursor and the speed of zooming depends on the distance from the camera center to the entity under the mouse cursor – getting to a building inside becomes very slow as the camera approach a facade through which we want to enter the building
- Use the camera options
  - it is hard to navigate in small/narrow areas (typically inside passageways) using standard combo pan + orbit mode, so using some kind of "look around" camera mode is better
- Use different face visualization styles
  - check correct orientation of faces (front/back) by coloring their sides in different color, switch to wireframe to identify unnecessary geometry (i.e. faces that are not visible from the outside, typically a wall/face between a two distinct buildings that touch each other)

6.6.5 Environment modeling

Even if the Prague Castle is the main focus of our virtual world, we cannot afford to ignore its surroundings. The environment around the castle create an atmosphere which is an integral part of the Prague Castle experience. A simple solution is the usage of sprites with photographs of the surroundings, cleverly placed in the areas from which the avatar is expected to see the surroundings. This method is very time and labor effective, and the result can be quite acceptable. It was used for example in the Virtual Old Prague project. However, since our application contains the FLY mode, this kind of trick would not work very well, because the avatar's movement is not restricted only to the ground.

For a convincing experience, we have to create the environment in 3D. The time extent of this diploma thesis does not allow for precise environment modeling, and since the environment is a complement of the application and not its main focus, we have settled for a way how to create visually acceptable environment in a reasonable time which involves the elevation data of the city. The elevation data is used for recreating a fairly precise 3D model of natural environment around the castle. There are several available source for such data. The most famous one is probably *The Shuttle Radar Topography Mission* (SRTM) [41], which was an international research effort that obtained digital elevation models (DEM) on a near-global scale to generate the most complete high-resolution digital topographic database of Earth. DEM has no universal definition, it is often used as a generic term for digital surface models (DSM) and digital terrain models (DTM). DEM is a continuous (raster) representation describing the shape of the surface where elevation is a function of latitude and longitude, and is mostly used as a generic term for [42]:

- Digital Surface Model (DSM)
  - describes a surface including buildings, vegetation and other objects
- Digital Terrain Model (DTM)
  - describes a pure terrain surface without buildings and vegetation in a way terrain elevation is given in topographic maps
  - can be obtained from DSM

The difference between DSM and DTM is depicted on Figure 6.8. The disadvantage of the mentioned SRTM data is a fact that it takes quite a while to find an exact place on Earth and the resolution of the data is not ideal for our purposes – it is still quite large. Fortunately, in April 2015, Geographic portal of the Prague city[1] published various geographical data related to the whole city. Among them is also a DTM in a form of a `*.TIF` image (see Figure 6.9), created from an aerial imaginery in 2010, with a map scale of `1:5000`. However, the image takes over 3GB, which complicates working with it, since conventional image editors cannot handle such big files. For this reason, the program called **OpenEV**[2] was used to display the image. OpenEV is a software library and application for viewing and analyzing raster and vector geospatial data and is capable of handling our DTM image.

We choose subarea in the DTM image with castle area roughly in the center and about 2 km of space around. Then, we export the subarea using OpenEV export method. If the option does not work, which is often the case, we must resort to taking screenshot

[1] http://www.geoportalpraha.cz

[2] http://openev.sourceforge.net

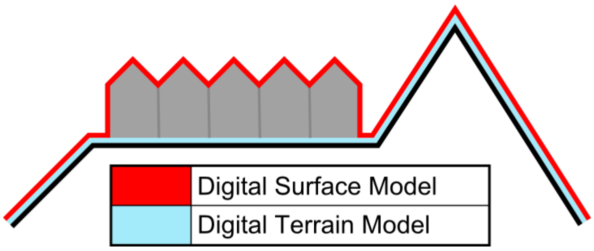


Figure 6.8. Difference between Digital Surface Model and Digital Terrain Model[43]

e.g. with `Print Screen` button on the keyboard (it may be necessary to take several screenshots and combine them together in an image editor).

Now that we have the image exported, which is essentially a heightmap, we can generate the terrain. We have used a program called **L3DT**[1]), which is similar to the popular Terragen[2]). The free version restricts resolution of the used heightmap to 2048×2048, but that proved to be enough for our purposes. We import the heightmap and set its parameters:

- Horizontal scale
  - smaller value = larger world, value of around 3 m is OK in our case
  - the program displays the heightmap along with the current map scale valid for the chosen value (in kilometers), thus it is easily possible to fine-tune the value so it roughly corresponds to the reality
- Vertical range
  - the lowest and highest altitude in the heightmap (in meters)
  - can be found using OpenEV which displays height information in meters for all pixels of the heightmap (175 m and 370 m in our case)

In L3DT, we can preview the resulting terrain from the heightmap in 3D to see if it looks acceptable (Figure 6.10 shows the heightmap used for generating the terrain and its render from L3DT; the red area in the heightmap represents the Prague Castle and the yellow triangle indicates viewing direction in the render image). It is also possible to create and interactively edit the texture of the terrain, which comes in handy. Finally, the process of incorporating the terrain into our virtual world has several steps:

- Import the terrain into the SketchUp
  - L3DT can export the terrain mesh into Collada format, which can be imported into SketchUp
- Correctly place the terrain and virtual world models
  - correct placement of the 3D models onto the terrain is tedious, but that is unavoidable
- Modify the terrain if needed
  - although we were careful during the heightmap creation, it is often the case that the terrain needs to be altered so the models fit
  - L3DT provides us with a set of tools with which we can edit the terrain interactively in 3D, and it works really nice

[1]) `http://www.bundysoft.com/L3DT`
[2]) `http://planetside.co.uk/products/terragen3`

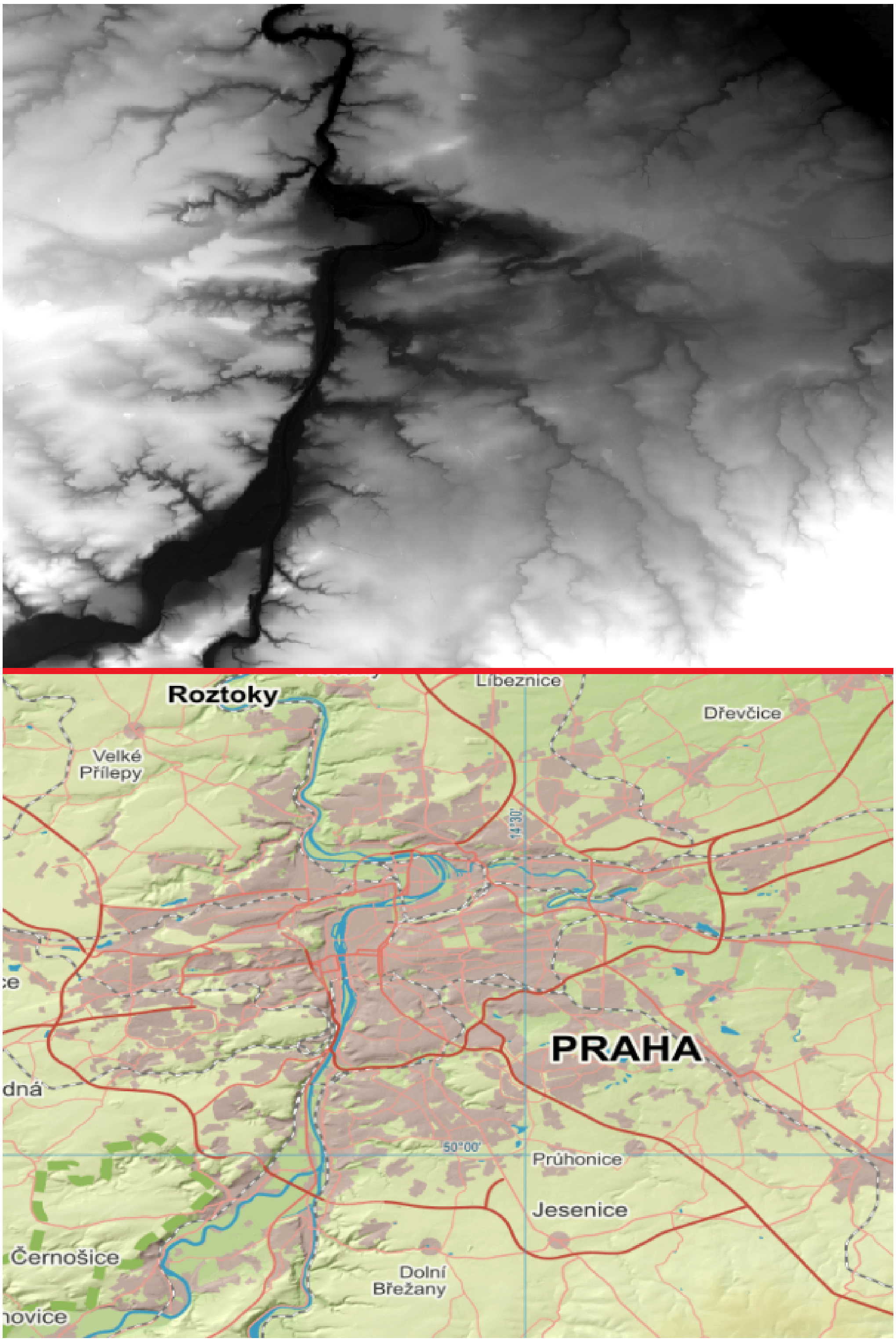


Figure 6.9. DTM[44] and corresponding classical map of Prague[45]

6.7 Exporting from SketchUp

We export the whole scene into Collada format (DAE) using standard SketchUp export functionality (`File - Export - 3D Model`). The exported file is then loaded into the application using the Collada loader bundled with Three.js.

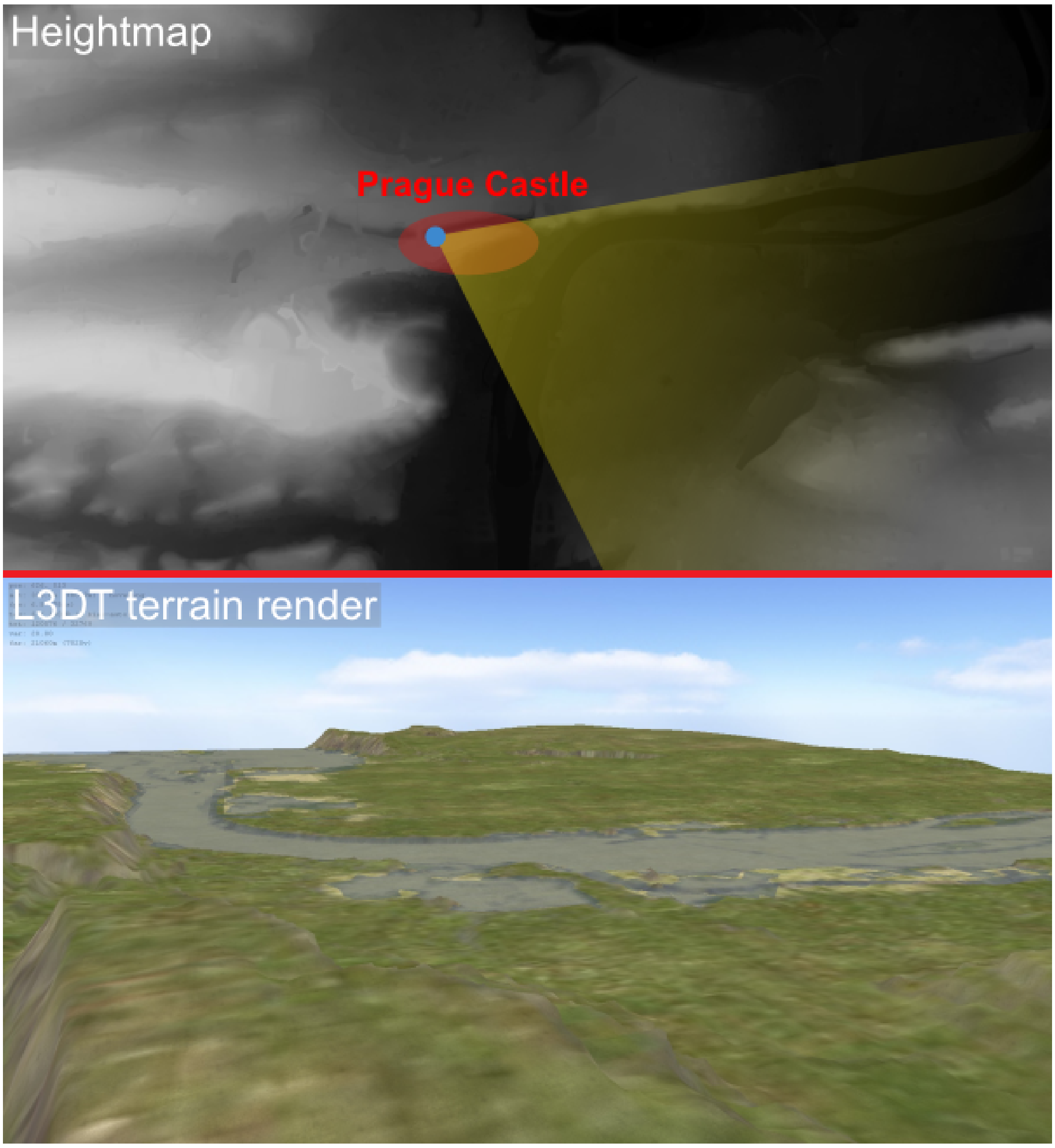


Figure 6.10. Terrain's heightmap and its render in L3DT

6.7.1 Issues with exporting

There is a major issue with exporting from SketchUp 2014 related to textures. When a texture is scaled/sheared/rotated during UV coordinates mapping (i.e. green, blue or yellow pins are being manipulated) on a face, which happens almost all the time, SketchUp exports the texture multiple times, in different versions – one new image file for each face which has UV mapping that involves a distorted texture. It goes without saying this is extremely inefficient and must be dealt with, otherwise it could cause a serious performance penalty for the application. The solution is to manually triangulate all geometry prior to the export. Although there is often an option of triangulating the geometry in the SketchUp built-in export dialogs, it does not solve the issue. Fortunately, a SketchUp plugin for UV mapping called **SketchUp UV**[1]) offers a function for triangulation which triangulates geometry in a way which solves the issue. With SketchUV activated, we select all geometry and execute the `Triangulate` option.

[1]) `https://extensions.sketchup.com/en/content/sketchuv`

It is necessary to inspect the scene afterwards because a few of triangulated faces may have now incorrect UV coordinates and thus must be adjusted manually. When we conclude that all triangulated faces are UV-mapped correctly, we can export the scene safely – SketchUp will not generate multiple versions of one texture anymore.

Another issue is related specifically to the export into the Collada format – faces are always exported as two-sided, regardless of what options we choose in the export dialog. We have decided to deal with this issue by a simple modification to the Collada loader for Three.js: since back-faces have no texture coordinates assigned, they are easy to identify and to ignore.

6.8 Statistics

There were 5 photographing sessions in the Prague Castle area, each taking 2,5 hours in the average. Almost 1500 pictures were taken (full area of the castle). Most of the sessions took place in the afternoon (2 P.M. to 8 P.M.) during the summer of 2015. Some of the building facades have different colors although its is evident that they should have identical colors, which is due to the fact that The Prague Castle is being continuously reconstructed and repainted.

Time spent creating one texture varies greatly (between 15 to 60 minutes), the average is about 30 minutes. Modeling and UV mapping of textures typically take a lot of time. For example, creation of the Matthias gate (for a screenshot, see Appendix C.8) and the passageway behind which goes to the Second Courtyard took around 10 hours of work; the balustrades on the rooftops of the First Courtyard took about 5 hours. The total time spent creating the virtual content is around 120 hours.

Total size of created textures on HDD is about 7MB, their total area is around 10 000 000px, which would make them around 40MB when uncompressed on the GPU. The whole exported scene with all models in Collada format and terrain mesh have around 12MB in total.

Chapter 7
User interface

The User interface plays an important role in visualization applications, since it attributes a lot to the overall experience. It is generally used as a mean to communicate different types of information, to change various options affecting the visualization, or even supply alternative controlling mechanisms for movement of the avatar. This chapter will discuss how the Visualization of Prague Castle coped with development of a suitable user interface which aims to accomplish these tasks. We will identify the target audience, the users' needs and important functionality for the user interface. The focus will not be just at the design decisions – the implementation specifics will be explained as well.

7.1 Goals

There are several things to consider before deciding on the course of the user interface (UI) design and its capabilities. It is vital that we **specify the target audience**, think about their motivation to use the application, and design an appropriate way for them to control it. These concerns can be summarized into the following questions:

- Who will use the application?
- For which purpose?
- How to make this easy for them?

Answer to the first question regarding **the target audience is simple: anyone**. Tourism and curiosity about historical places is a thing which can be shared across people of all age and origin. Thus, application aims to provide an experience which will benefit both teenagers and those of our kind who were born even much earlier in the previous century. To broaden the target audience, we have decided that the **application will be available in two languages**, with a possibility to switch between them during the runtime:

- Czech
  - official language of the country where Czech Technical University and the Prague Castle are located
- English
  - one of the most widespread languages
  - especially prevalent in the information technologies and web in general

Purpose of visiting virtual places are consequent upon the already mentioned tourism and historical curiosity – users want to know more about the place, and in this case, mainly in terms of the visual experience. However, it must not be forgotten that when they encounter some interesting object during their virtual walk, they may want to know more about it, perhaps read a small article or obtain a web address where full info with all exhaustive details can be found. It follows that our **visualization shall contain a mechanism which allows user to educate themselves about the environment or interesting objects in a further detail**.

Since the users of the application are expected to have a varying computer skill and experience with similar applications, it was decided that the user interface would be rather simple, so even an inexperienced user will not be feeling lost and controlling the application would be easy for him/her. **Some kind of help shall be available to the user at all times to prevent a confusion**. We want to point out that mainly people working in the 3D graphics field and gamers are able to navigate freely in the 3D. Regular users who are not accustomed to the virtual environment are not usually very proficient in controlling of the avatar in the standard way (i.e. using the keyboard or mouse), but they seem to pick on the **navigation using on-screen buttons** [46] and welcome **lists of simply accessible viewpoints**. We shall keep in mind these kind of users when designing the user interface. Navigation could be also made easier for them by adding a **2D map synchronized with the user's movement** as suggested in [33]. We have decided to take this one step further and **include a big map** (resembling a classical map) as well with marked positions of viewpoints and such. Taking our conclusions from Chapter 2 into the account, we have decided to include possibility of **flying with the avatar**; a mechanism for switching between regular virtual walk and flying will have to be created. More advanced users may want to **change settings of the application** (e.g. visual effects settings) and curious users may want to know who is responsible for the application, so we shall somehow allow them to do these tasks e.g. by providing some kind of menu.

To summarize the ultimate goal and the most important points – we want the interaction between the user and the application to be easy and provide the user with functionality expected from a virtual walk application. This means:

- create the user interface that will be understood by the entire target audience
- provide the users with help and navigation means within the virtual world
- allow the users to educate themselves about interesting objects in the virtual world
- allow the user to edit settings of the application

7.2 UI functionality

Based on the defined goals, a simple task analysis was performed. The hierarchical task analysis diagram depicting these is shown on Figure 7.1. The explanation of the provided functionality follows.

Edit settings is about changing application behavior and appearance. Settings are split into the groups depending on their type:

- General settings
  - control options (mouse sensitivity, ...)
- Visual settings
  - changing lighting conditions and turning effects on/off
- Language settings
  - switching between Czech and English

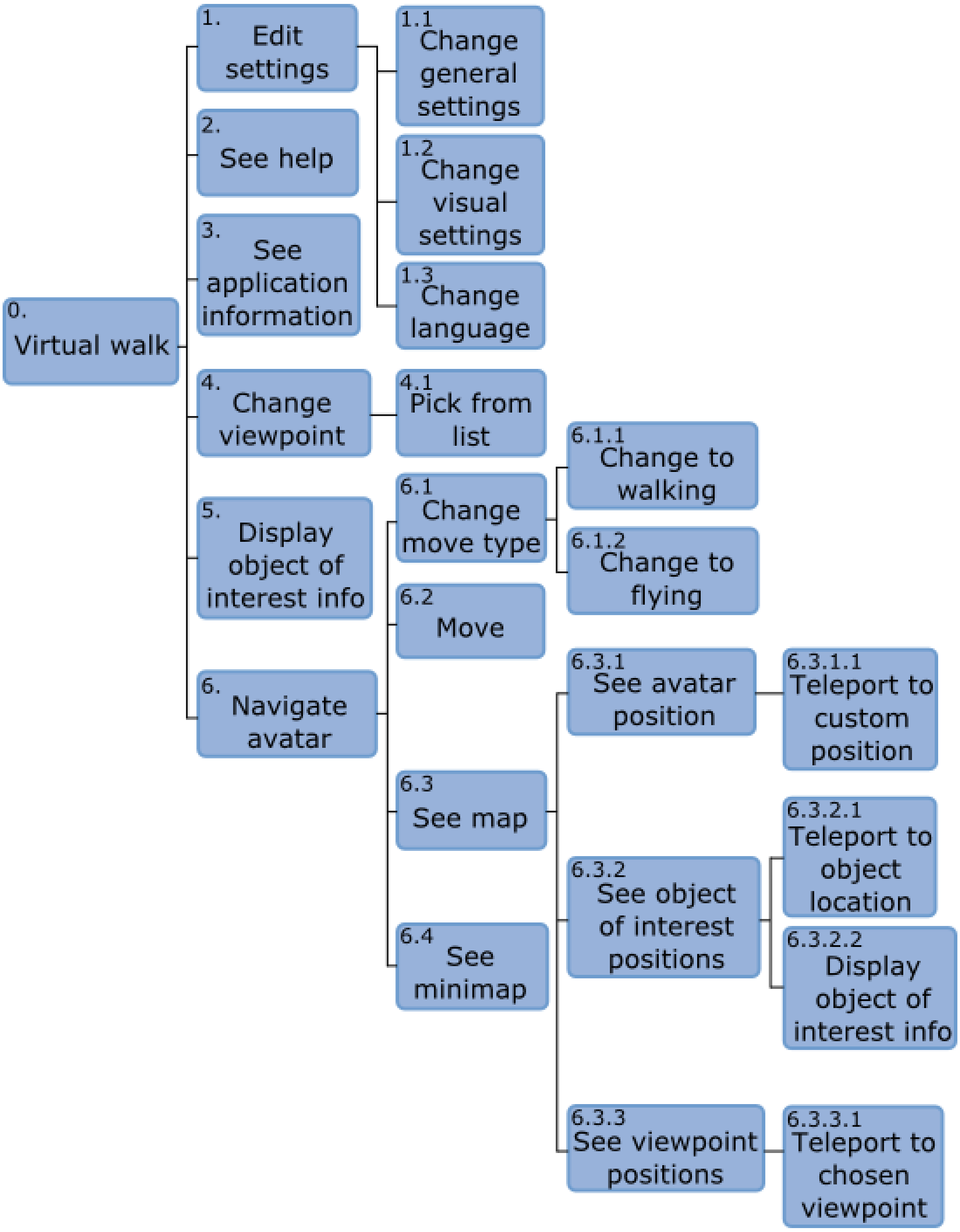

0. Virtual walk
1. Edit settings
1.1 Change general settings
1.2 Change visual settings
1.3 Change language
2. See help
3. See application information
4. Change viewpoint
4.1 Pick from list
5. Display object of interest info
6. Navigate avatar
6.1 Change move type
6.1.1 Change to walking
6.1.2 Change to flying
6.2 Move
6.3 See map
6.3.1 See avatar position
6.3.1.1 Teleport to custom position
6.3.2 See object of interest positions
6.3.2.1 Teleport to object location
6.3.2.2 Display object of interest info
6.3.3 See viewpoint positions
6.3.3.1 Teleport to chosen viewpoint
6.4 See minimap

Figure 7.1. Hierarchical task analysis diagram for the user interface

See help displays the information about the virtual walk controls and explains to the user what is possible in the application and how to achieve it.

See application information displays the information about the application itself – what is it, who made it and why, and provides means of contact to the involved people.

In **Change viewpoint**, the user is presented with a list of available interesting locations in the Prague Castle area. By choosing some of the viewpoint, avatar is instantly teleported to the viewpoint location.

Display object of interest info presents the user with a short description and historical information about objects.

Navigate avatar comprises of:

- Switching back and forth between the walking and flying modes
- Moving the avatar in the virtual world
- Maps for easier orientation
  - Big map
    - an interactive map of the Prague Castle area, resembling a classic paper map
    - aims to help the user with the orientation
    - locations of the avatar, available viewpoints and objects of interest are marked there
    - possible to choose a custom position and teleport the avatar there
    - the avatar can be teleported to the locations of viewpoints and objects of interest as well
    - also possible just to display the information about any of the objects of interest
  - Minimap
    - smaller version of the Big map with less details and no interactivity
    - for quick orientation in the area
    - intended to be permanently visible

7.3 Implementation, design and UI behavior

7.3.1 Technologies and libraries

We have taken an advantage of the fact that Visualization of Prague Castle is a web application and used HTML, CSS and JavaScript to create its user interface. The UI layout (i.e. positions of buttons, text size, etc.) is defined via CSS using relative positioning and relative size units, which makes the design flexible; it keeps its intended look on a vast number of resolutions, making the application suitable not just for large monitor screens, but also for a lot of smaller displays (mobile phones, tablets, ...).

jQuery UI[1]), a library which provides many visual widgets, is the main cornerstone of the UI. The basis for color design was a jQuery UI provided theme called "Eggplant", a rather dark theme with shades of gray, blue and violet. Styling of the theme was customized from both visual and coding perspective.

Colors were adjusted, background images were replaced and pure colors were used instead (the background images did not look very good when the application was in either very small or very big resolution). Each used image is in the SVG format (images in other formats from libraries such as jQuery UI or similar were manually recreated as SVG). These modifications ensure that scaling for different resolutions will not cause any visual quality degradation. According to [47], SVG format is supported by all current mainstream browsers, so there is no risk that the images will not be displayed on some configuration.

The most used jQueryUI widget in our UI is the "`Dialog`". This widget can create a window (implemented as a `<div>` container in pure HTML/CSS/Javascript) which

[1]) `https://jqueryui.com`

resembles a standard window appearing in an operating system, i.e. it has titlebar, closing cross icon in the upper right corner, body with text and possibly some buttons at the very bottom (e.g. OK, Close). However, its default styling had (again) problems with scaling into small/big resolutions and some visual elements did not fit into the intended application UI style, so its style has been heavily modified. Functionality has been extended as well to add support for multiple languages.

Menu window, although looking similar to windows created with the `Dialog` widget, is implemented using the modified jQueryUI "`Tabs`" widget which provides functionality similar to tabs (panels) in web browsers. Both menu window and every dialog window contain three buttons for manipulation with a window:

- Return / OK
  - closes the window (and possibly returns to the virtual walk)
- Language switch
  - switches application to another language
- Close cross button
  - closes the window (and possibly returns to the virtual walk)

The goal which we had in mind when putting language switch button to the most of the windows was that the button will be visible at all times. Application chooses its language automatically based on the language settings of the user browser, but nevertheless, it is important to provide the user with a clearly visible option of switching between the languages.

The main part of the user interface is a head up display (HUD), which is always visible (Figure 7.2). HUD is displayed in the lowest area of the screen, near the bottom border, and is split into the three parts:

- left
  - buttons for Menu, Help, Walk/Fly switch
- middle
  - Virtual keyboard for moving
- right
  - buttons for Viewpoint change, Map display
  - Minimap

The buttons have circular shape, with an icon inside (symbolizing their purpose) on the partly transparent dark background. Each button also has its name written above it so the users do not have to wonder what the button is for. Buttons' colors are changed slightly when the user hovers over them.

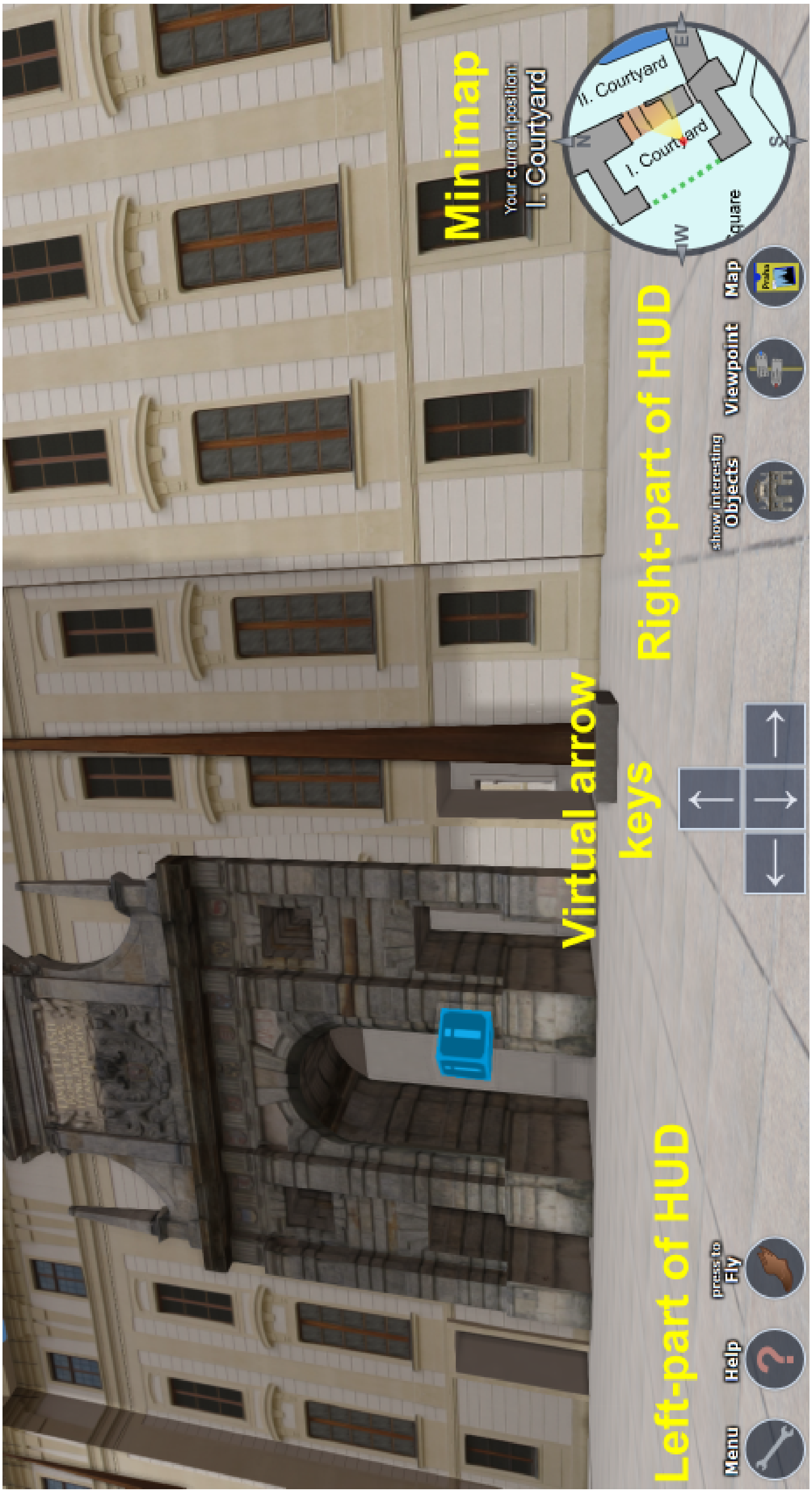


Figure 7.2. HUD in the application

7.3.2 Left part of HUD and virtual keyboard

First button on the left part of the HUD is for opening the application Menu, which carries out **See application information** and **Edit Settings** functionality mentioned in the previous section. The menu has 4 tabs (categories):

- General settings
    - mouse sensitivity and orientation of Y axis (normal vs inverted)
    - left and right arrow moving style (turning the avatar's head vs moving sideways, i.e. strafing) when using keyboard
    - collision detection
- Display settings
    - turning visual effects on/off
- Light settings
    - changing the current time of the day for the day/night simulation (more on this in Chapter 10)
    - switching lights in the castle area on/off
- Credits
    - information about the application and the thesis
    - contact to the author and supervisor of the thesis

Second button in the left part of the HUD is for opening the window with **Help**. Help section contains information about controlling the whole application, explains what is possible and how to achieve it. Especially controls (keyboard, mouse, buttons in the user interface) are explained carefully and with images for better and faster understanding – see the User Manual in the appendix E.12.

Third button in the left part of the HUD is for **changing avatar moving state between walking and flying**.

Virtual arrow keys are placed in the middle part of the HUD in order to draw the user's attention. They are clickable and serve as **an alternative way of controlling the avatar**, since less experienced users may not be accustomed to moving with the real keyboard arrow keys. On top of that, if the user tries to access an inaccessible location (e.g. tries to walk forward in spite of the fact that a wall is in front the user), the arrow keys are drawn in red shades, signalizing that the further movement in the current direction is not permitted.

7.3.3 Viewpoint change

The first (leftmost) button in the right part of the HUD is for **Viewpoint change**. When the button is clicked, the window with available viewpoints is opened (Figure 7.3). Available viewpoints are arranged in a grid and are represented by an image (with their name written above it). We have chosen this approach to help the users with their decision – seeing viewpoint images makes viewpoints less anonymous and the user can decide which viewpoints interest him/her the most.

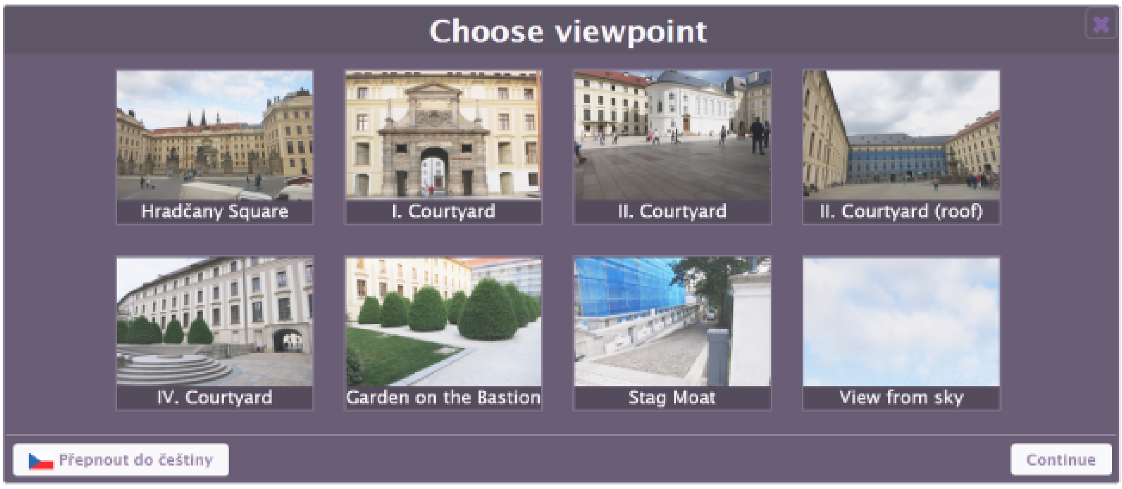


Figure 7.3. Window with the list of viewpoints

7.3.4 Map

Second button in the right part of the HUD is for displaying a window with the interactive **Big map**. Lets briefly discuss the process of the map image creation. There are generally two options when it comes to retrieving/displaying a 2D map:

- use a map provider
  - Google Maps, OpenStreetMap, ...
- use an own map

It was decided that an own map will be used. Integration of a map from providers in the desired form into the application is not trivial and there would be problems with synchronization of position in the virtual 3D world and provided map, because the virtual world does not match reality to the 100 %. And since The Visualization of Prague Castle takes place on a rather small area, it is more efficient to use an own map instead, which can be customized visually and its implementation is also straightforward.

The process of creating the own map is manual, but not complicated. Previously captured heightmap (will be discussed in the future chapters about collision detection, namely in 8.3.1) image is used as a template for creating the map. This ensures that the map image corresponds 100 % to the virtual world. The heightmap image[1]) looks very much like a regular map, except for the colors. A picture is worth a thousand words, so see Figure 7.4 where the real-world orthophoto map and our application's heightmap are drawn side-by-side[2]). Now that we see that the heightmap is not good just for the collision detection, but also for displaying the map of the area, the goal is to change the colors and also have the map in a vector format, which allows scaling without visual degradations. This can be achieved with a vector image editor, e.g. Inkscape[3]). Using a vector image editor, a new SVG image with the same dimensions as the heightmap image is created, and vector outlines of buildings and important objects are drawn (the heightmap image is displayed on the background, we draw "over" it). It is necessary

[1]) When we talk about the heightmap, we mean the "Largest height layer" if not specified otherwise (explanation will be provided later in 8.3.3)

[2]) The heightmap matches the orthophotomap so well because we have been using aerial imaginery when modeling the outline of Prague Castle (to get the building shapes and distance correctly)

[3]) `https://inkscape.org`

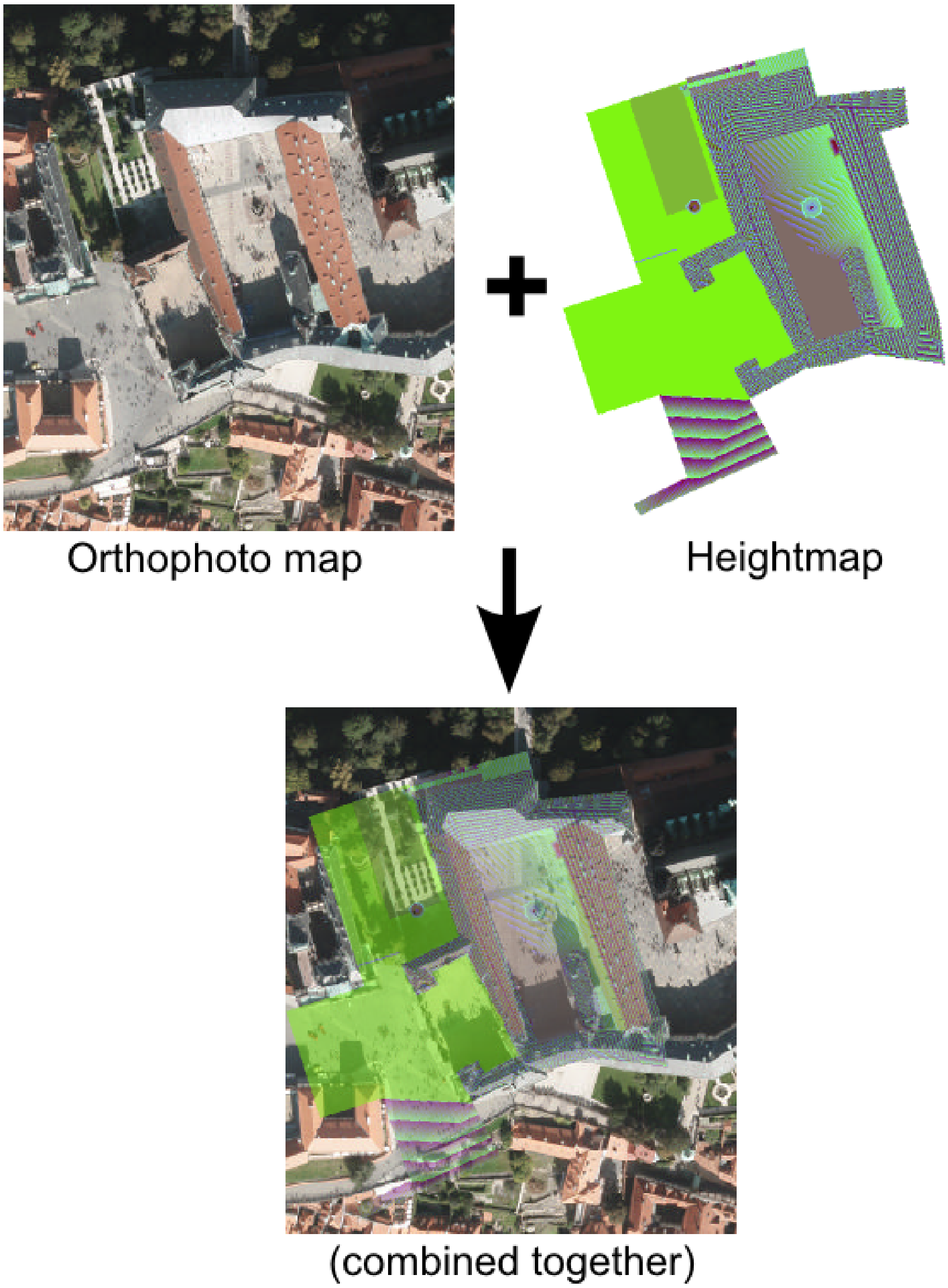


Figure 7.4. Real-world orthophotomap and heightmap – proof that they match together and thus heightmap can be used for creating a map of the area

to strictly respect edges of objects, i.e. do not alter their positions. When this step is finished, we have basis of our map.

In the next step, the map (currently containing only outlines of objects) is made more visually appealing by addition of colors. Areas in the map which represent an object of interest are colored in a way which especially stands out, and each such area is given an ID in a vector image editor – later, when an area is clicked/hovered over in the application, we know to which object of interest it relates to using this ID.

Last step is the placement of location/street names (e.g. "The First Courtyard", "Garden on the Bastion") into the map in a form of text captions. SVG Text element[1]) is used as a caption, and just like with objects of interest areas, each such element has an ID assigned. The ID is used for the name translation purposes – it would not be possible to translate location/street names without it when switching the languages (the translation mechanism will be discussed later in 7.3.7). Actually, it does not matter which text is written in SVG Text elements in the map image (since this text will be overwritten by the application's translation mechanism), all that matters is that SVG Text elements are placed correctly in the image and have correct ID. However, we recommend to write location names into SVG Text elements in the image as well, because then it is easy to spot if there is enough space for the name or not and in that case the element should be moved a little bit.

When the area related to an object of interest is clicked/hovered over, a small window with name and an image representing the object of interest appears, presenting the user with two options (as can be seen on Figure 7.5, which depicts the Big map dialog):

- Go to location
  - teleports the avatar to the location of the object of interest in the virtual world
- Open object information
  - displays window with text information about the object of interest

Locations of the available viewpoints (i.e. those from the **Change viewpoint** window) are marked in the map as well, in a form of icons (highlighted on Figure 7.5 in red). Just like with areas representing objects of interest, these icons can be clicked/hovered over and a small window similar to one described in case of objects of interest appears, though this time, it contains only the option of teleporting to the viewpoint's location. Icons representing viewpoints are added into the map during the application runtime and their exact placement in the map is computed from 3D positions of the corresponding viewpoints (using projection from 3D to 2D – inverse process to computing 2D map position from the avatar's 3D position which will be discussed later in 8.3.2).

The avatar's current location is represented as a sticky-figure (similar to Google Maps). The figure can be dragged and dropped around the map – avatar is teleported to the drop position (but only if the position is accessible, i.e. no teleporting outside of the castle area is allowed).

7.3.5 Minimap

The last element in the right part of the HUD is the **Minimap**, which is intended to visually resemble a compass; that is why there are also cardinal directions marks N, S, E, W. A part of the Big map corresponding to the area in the immediate distance from the avatar is displayed inside the compass. This effect is achieved by using CSS property `overflow: hidden` along with a translation of the map image position. Avatar's position and viewing direction is symbolized by a red dot and a yellow triangle. Name of the current location is displayed above the Minimap. Finding out the location name is simple: we have an image where locations of the areas are drawn in different colors (we call this the *"Area map"*) which is derived from the (height)map, as we can see on the example on Figure 7.6. We look at the color of the pixel at the avatar's position in the Area map and based on that color, we know where the avatar is (no color means that an area is not accessible, everything else corresponds to some area).

[1]) `http://www.w3.org/TR/SVG/text.html`

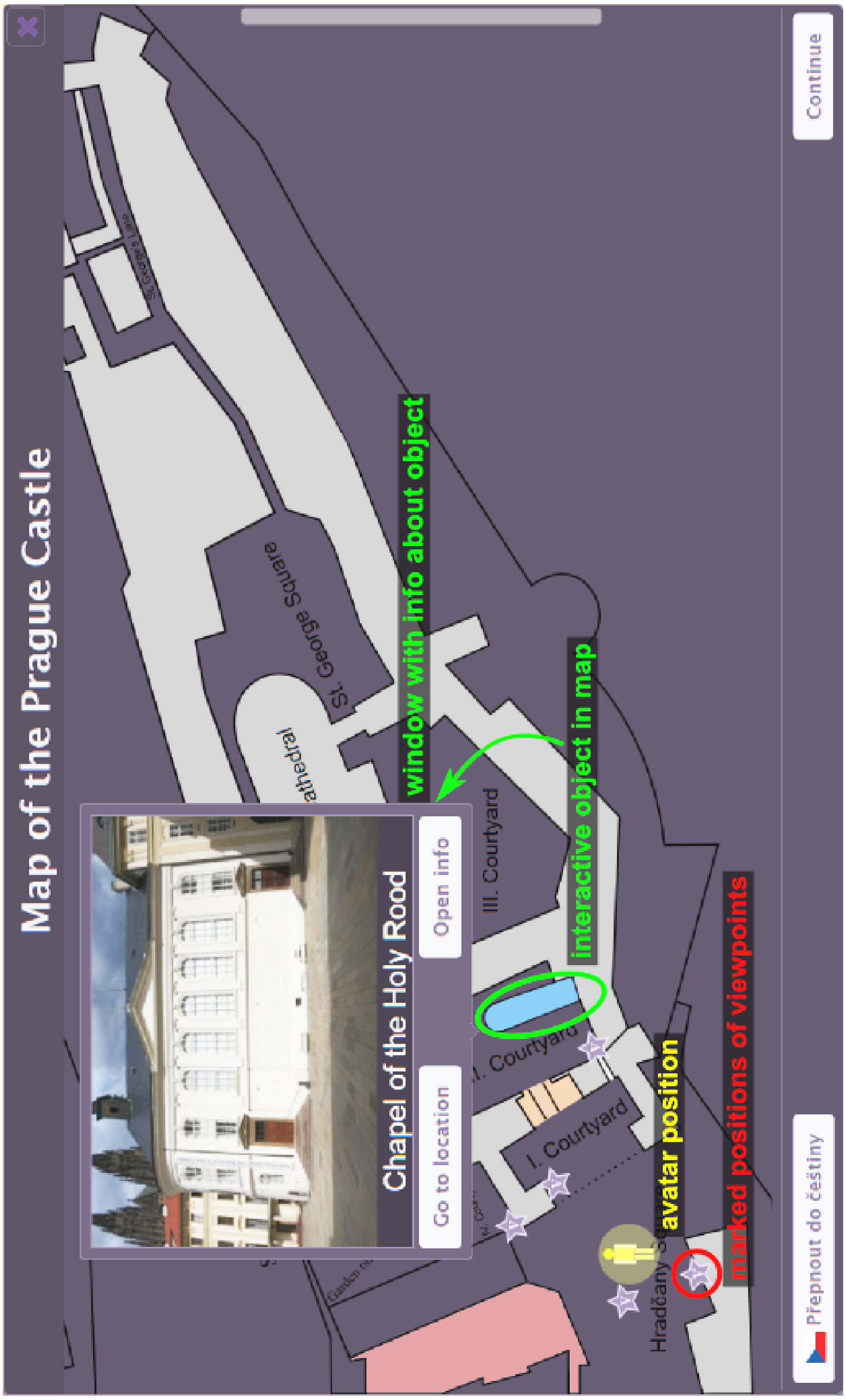


Figure 7.5. Big map

7.3.6 Interaction with objects of interest

The interaction with interesting objects could be done from the Big map as was discussed earlier in 7.3.4, but it is more practical to interact with objects during the walking itself – by hovering/clicking on an object of interest in its 3D representation. And because it may not be apparent which objects are meant for interaction unless the user hovers the mouse over them, we have decided to put a small 3D "info" spinning

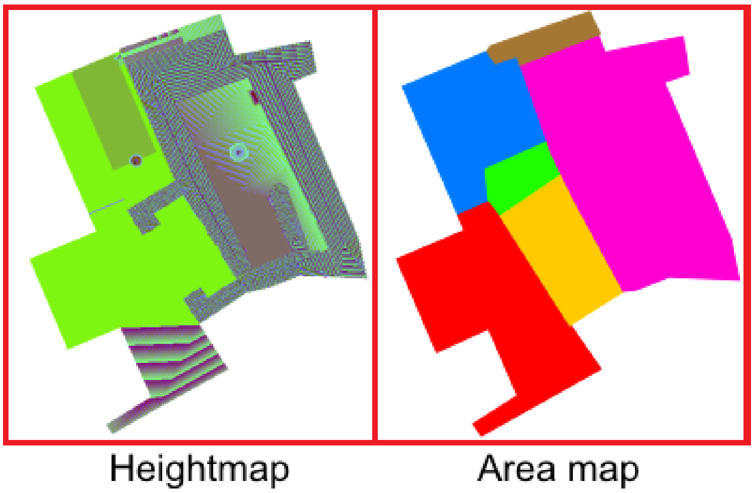
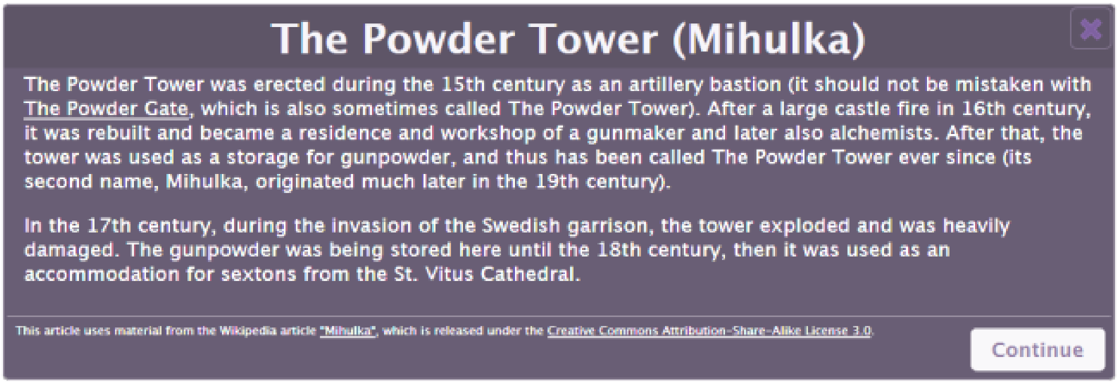


Figure 7.6. Area map

cube near all of them to suggest a possibility of an interaction. If the user hovers over either an object or an info cube associated with it, the name of the object is displayed near the mouse cursor; if the user clicks, the article for the object of interest is opened (see Figure 7.7).



The Powder Tower (Mihulka)

The Powder Tower was erected during the 15th century as an artillery bastion (it should not be mistaken with The Powder Gate, which is also sometimes called The Powder Tower). After a large castle fire in 16th century, it was rebuilt and became a residence and workshop of a gunmaker and later also alchemists. After that, the tower was used as a storage for gunpowder, and thus has been called The Powder Tower ever since (its second name, Mihulka, originated much later in the 19th century).

In the 17th century, during the invasion of the Swedish garrison, the tower exploded and was heavily damaged. The gunpowder was being stored here until the 18th century, then it was used as an accommodation for sextons from the St. Vitus Cathedral.

This article uses material from the Wikipedia article "Mihulka", which is released under the Creative Commons Attribution-Share-Alike License 3.0.

Figure 7.7. Window with an article about the object of interest

From the implementation standpoint, it is necessary to be able to find out object which is currently under the mouse cursor. The process which determines this is called the **picking**. The solution used for solving the picking problem is a common one, described for example in [48]. The main idea is that each pickable object has assigned different ID, which maps to a unique color (the mapping is bijective). The scene is rendered to an off-screen buffer (we will refer to this as the *"pick draw"*), with only pickable objects being drawn in their unique colors. Figure 7.8 shows the difference between the scene rendered normally and in the pick draw[1]) (in this example, the fountain and the chapel are the only visible pickable objects). Then, color of the pixel which is currently under the mouse cursor is read – knowing color of this pixel, we know which object was picked (if any), since each object has a different color assigned. From the implementation reasons, we have to convert each (integer) ID to a (float) number

[1]) Actually, the real pick draw in the application would probably not look "pretty" like this with such distinguishable colors for eyes, but that is not important for the correctness of the algorithm.

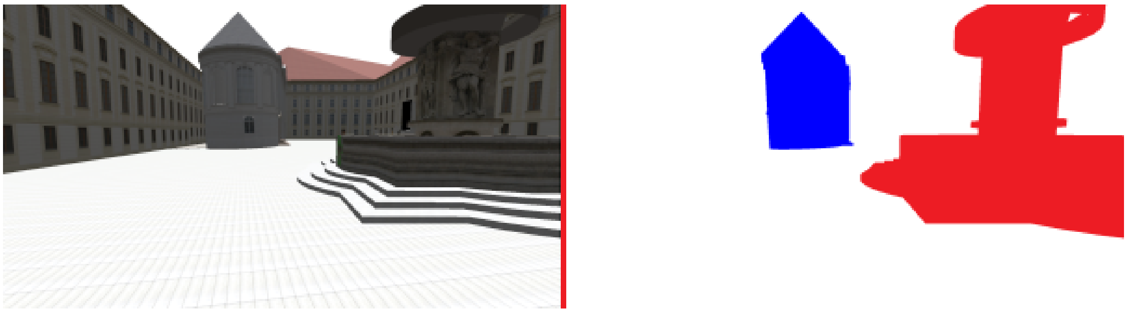


Figure 7.8. Regular draw vs Pick draw

between 0 and 1 – details of encoding a number into a color will be explained further in Chapter 8.3.1 regarding heightmaps.

The pick draw is not performed on each frame or mouse move – reading its pixels that often would slow the application significantly down. However, the application absolutely needs the information about object under the mouse, at all times. The solution for this is easy; the pick draw is performed and its pixels are retrieved every time when (after) the view is changed. So until the view changes again, off-screen buffer with the picking information is up-to-date and it is possible to read the value of the pixel under the mouse right away. The problem is when the avatar is changing its position (i.e. when walking), because then the view changes continuously. This is why a small delay (about 500 ms) between the view change and the pick draw with pixels reading has been introduced. In other words, the pick draw will be executed only if the view has not changed in the last 500 ms. Because the delay is so small, the user is not capable of detecting the fact that he/she is not able to interact with objects for these first 500 ms when the view changes.

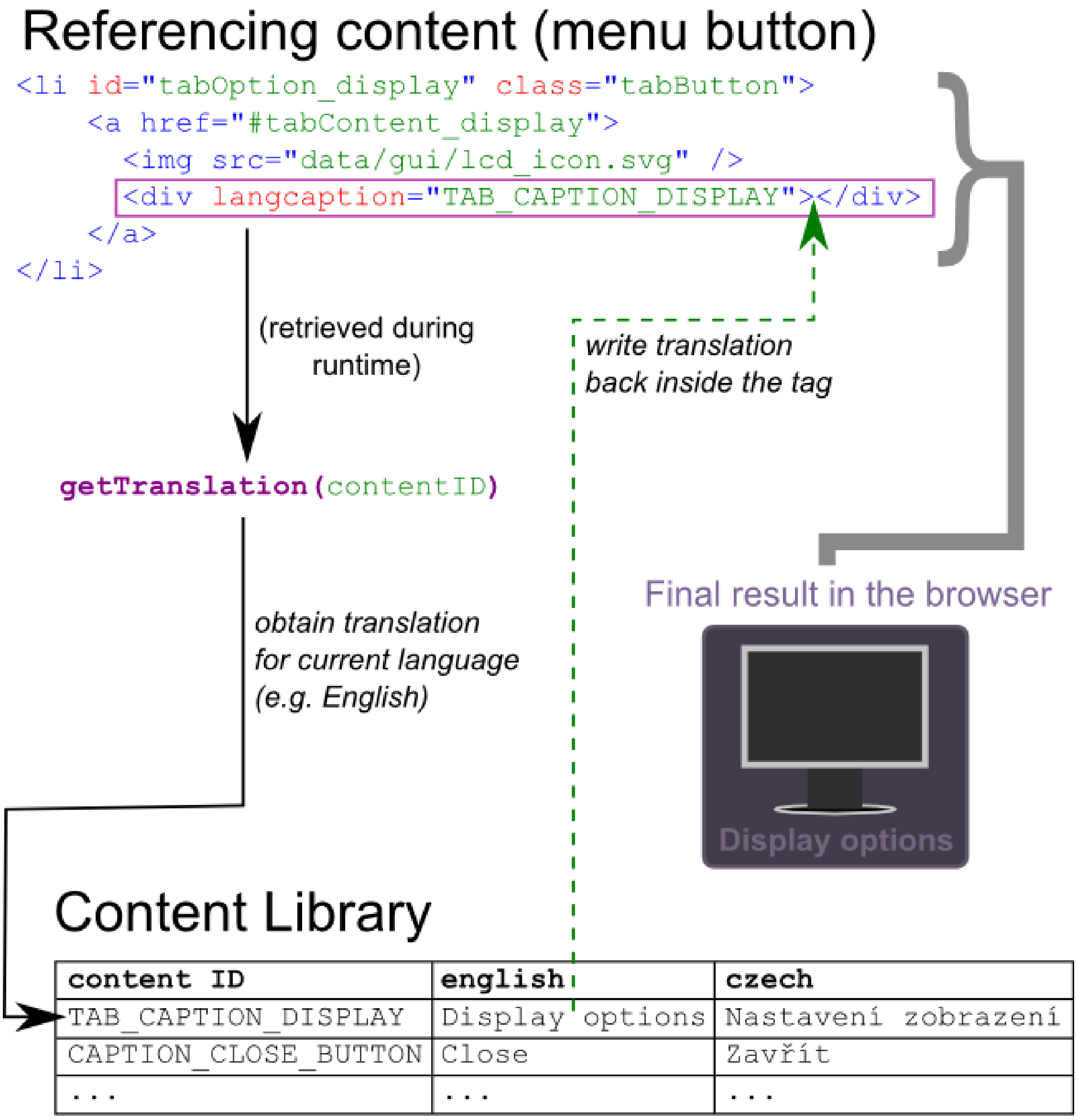
7.3.7 Translation mechanism

Since the application supports multiple languages, with an option of switching between them during the runtime, we had to find a way how to implement this feature. We set following requirements for the implementation:

- speed
  - switching between the languages should be fast (i.e. application should not restart itself), the user should not have to wait for the translation of a text
- practical separation of presentation and content
  - we want to be able to change how the content is displayed without affecting the content itself
  - we want to prevent the code duplication
  - but at the same time, we do not want to sacrifice practicality just for the sake of the separation

Naive solution would be to have different HTML pages for each language. We would end up with several HTML pages for every article, menu etc. Then, for example changing visual appearance of one caption, adding a paragraph, or modifying some sentence would require editing of several HTML pages – that is very impractical and chaotic, code is heavily duplicated. This is problem especially in case of pages related to the UI (menu, dialog windows, ...), because they contain a lot of HTML markup tags around.

In our solution, we use a special class which acts as a content library: we give each piece of content (i.e. caption, sentence, ...) unique string ID and we are able to store translations for different languages along. It basically works as a table. Because the application does not contain big amounts of text content, we have defined content ID's and their translations just in the special source code file, but obviously the application can be extended to read text definitions and translations e.g. from a XML file if needed. When we want to use a piece of content in a HTML page, we add the special attribute "`langcaption`" which references the content's ID to a `<div>` tag which is placed where we want the content to appear. During the runtime, the application inserts the correct translation of such content piece right to the place where the tag is. This allows us to have a single HTML page for all languages (no code duplication) and since the content is stored separately, it is easy to modify it. The solution flow is illustrated on the example of a main menu button definition and can be seen on Figure 7.9.



Referencing content (menu button)

```
<li id="tabOption_display" class="tabButton">
    <a href="#tabContent_display">
      <img src="data/gui/lcd_icon.svg" />
      <div langcaption="TAB_CAPTION_DISPLAY"></div>
    </a>
</li>
```

```
getTranslation(contentID)
```

(retrieved during runtime)
write translation back inside the tag
obtain translation for current language (e.g. English)
Final result in the browser
Display options

Content Library

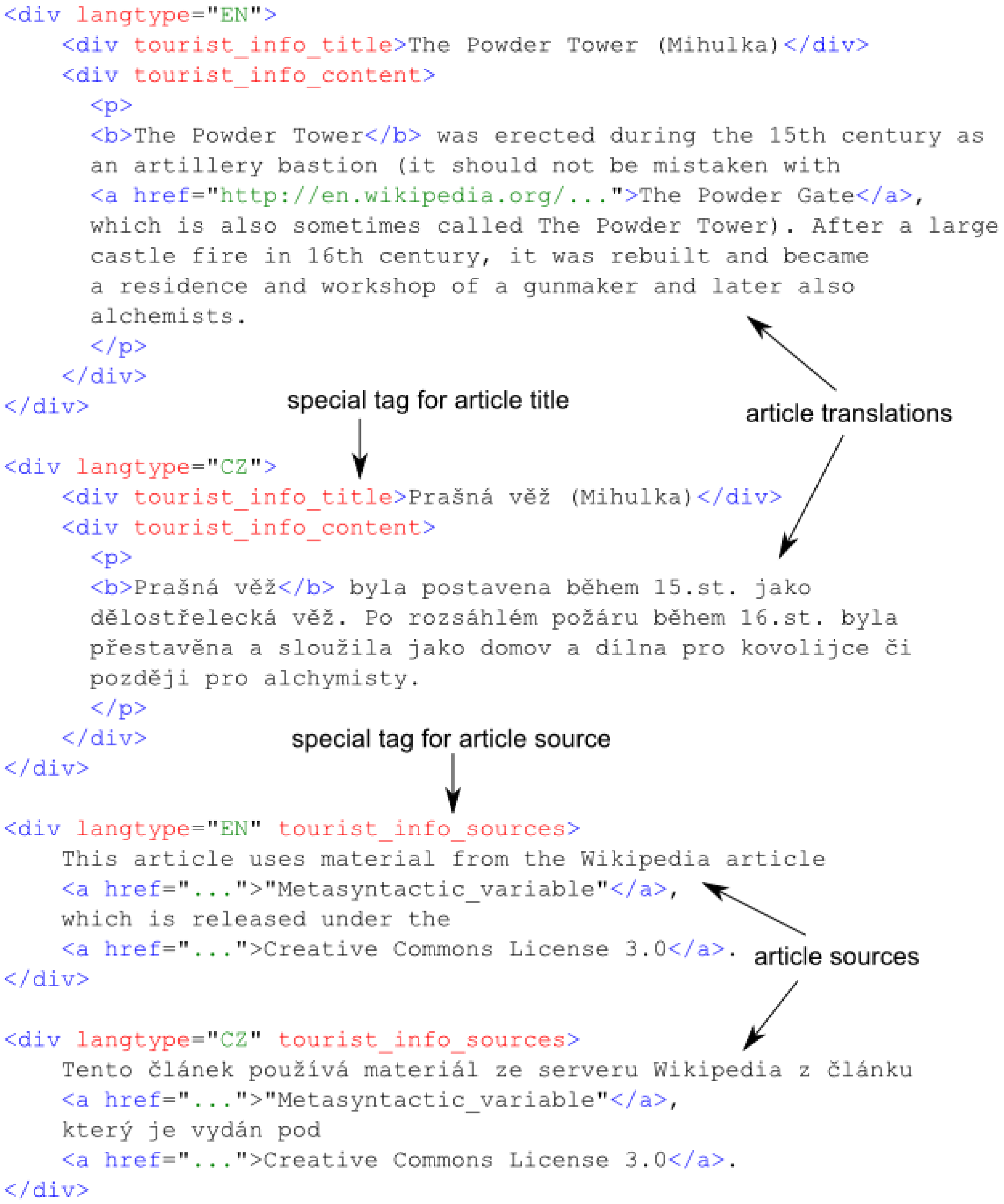
content ID	english	czech
TAB_CAPTION_DISPLAY	Display options	Nastavení zobrazení
CAPTION_CLOSE_BUTTON	Close	Zavřít
...	...	...

Figure 7.9. Content translation mechanism

However, assigning an ID to everything is not very practical. While there is no problem with captions for the UI generally or with short text paragraphs, the situation changes when we want to present longer articles, with a possibility of a different formatting for each article, including images etc. This is the case of articles with information about some object of interest. For the sake of simplicity and practicality, content from these articles has no ID assigned (is not present in the content library). The content is stored in separate HTML pages (one page per article), each page includes translation of content for all languages. This unfortunately introduces the problem that when we want to change an appearance of one paragraph, we have to make this change in all other corresponding paragraphs written in different languages, luckily the good thing is that apart from some formatting tags there and here, there is no excessive markup (which is common in pages related to the user interface) so it is not much of a hassle, which makes this downside acceptable. A very simple example of such article with its translation HTML page (some parts of the page were omitted) is depicted on Figure 7.10 (how this exact article looks like in the application was shown on Figure 7.7). There are four attributes used in articles' HTML pages to mark different parts of an information within the page:

- `langtype`
  - marks that everything inside its tag is related to the language specified by the attribute
- `tourist_info_title`
  - marks the title of an article
- `tourist_info_content`
  - marks the translated article content (information about an object of interest)
- `tourist_info_sources`
  - marks translation of the section that lists sources used during creation of an article

As for the requirement of the fast translation speed: since the content is referenced in HTML tags, the page's document object model (DOM) has to be traversed in order to find those tags and put correct translations inside them. We want to avoid a potentially slow DOM traversal each time when the languages are switched. This is the reason why for each page, we traverse the DOM only once and store found HTML tags with content references into the special list. When the languages are switched, we retrieve those lists for all pages which are currently visible and put translations into all stored tags, which is fast.



```
<div langtype="EN">
    <div tourist_info_title>The Powder Tower (Mihulka)</div>
    <div tourist_info_content>
      <p>
      <b>The Powder Tower</b> was erected during the 15th century as
      an artillery bastion (it should not be mistaken with
      <a href="http://en.wikipedia.org/...">The Powder Gate</a>,
      which is also sometimes called The Powder Tower). After a large
      castle fire in 16th century, it was rebuilt and became
      a residence and workshop of a gunmaker and later also
      alchemists.
      </p>
    </div>
</div>

<div langtype="CZ">
    <div tourist_info_title>Prašná věž (Mihulka)</div>
    <div tourist_info_content>
      <p>
      <b>Prašná věž</b> byla postavena během 15.st. jako
      dělostřelecká věž. Po rozsáhlém požáru během 16.st. byla
      přestavěna a sloužila jako domov a dílna pro kovolijce či
      později pro alchymisty.
      </p>
    </div>
</div>

<div langtype="EN" tourist_info_sources>
    This article uses material from the Wikipedia article
    <a href="...">"Metasyntactic_variable"</a>,
    which is released under the
    <a href="...">Creative Commons License 3.0</a>.
</div>

<div langtype="CZ" tourist_info_sources>
    Tento článek používá materiál ze serveru Wikipedia z článku
    <a href="...">"Metasyntactic_variable"</a>,
    který je vydán pod
    <a href="...">Creative Commons License 3.0</a>.
</div>
```

Figure 7.10. HTML page with a translated article

Chapter 8
Collision detection

Since the virtual world is an imitation of the real world, a virtual walk application should aim to mimic its important characteristics in order to make the experience "believable". One of the most important characteristics of the real world relevant to virtual walks is the impact of physic laws. This chapter deals with simulating physical phenomena related mainly to walking and gravitation. Our simulation reduces to the problem of collision detection between the user's avatar and the virtual world. We will discuss several collision detection approaches, evaluate their pros and cons, and describe implementation of the chosen approach and incorporation of the gravitational law into it.

8.1 Goals

Our goal is to simulate the most common physical phenomena that affect a position of a human body during walking, but we also want to take an advantage of a virtual environment and simulate phenomena which are not likely to happen in the real world, such as flying.

We shall consider these physical phenomena in our simulation:

- walking on an accessible terrain
- collisions with objects
- gravitation

Let us define them.

Walking on an accessible terrain is a phenomenon where the avatar's imaginary leg position copies a terrain, i.e. their position is the same as the height of the terrain at particular place in the world. The decision if the place on the terrain in the virtual world is accessible or not should correspond to the expectation one would have if trying to walk on such terrain in the real world.

Collision with objects phenomenon should prohibit avatar to pass through solids, e.g. it should not be possible to walk through walls, statues and such.

Gravitation phenomenon should resemble Newton's law of universal gravitation [49] under an assumption of constant gravitational attraction for a falling body near the surface of Earth. In other words, if the avatar ends up in a situation when it is in the air (e.g. when avatar walks/"slips off" down from a roof), it will fall with an increasing speed and eventually stop falling when it lands on the nearest surface below, usually the ground.

We should be able to **turn simulation of the each mentioned phenomenon on or off** anytime. Imitating the real world is important, but simultaneous option of not imitating is where virtual walks can really shine, giving the users a way how to go beyond the physical limits of the real world through their virtual avatars. Probably the most interesting usage of turning phenomenon simulation off is in case of the gravitation, which enables avatar to fly, allowing for fly-through over the virtual world or just getting to normally inaccessible locations such as buildings' roofs.

Since all named physical phenomena in our simulation reduce to the problem of collision detection, we shall decide on the method of the collision detection for our application and implement it. It was decided that the **collision detection computation demands should be as little as possible**. We think there is no need for perfect simulation in our application – we are willing to trade precision for speed and aim to create an illusion which is just good enough.

8.2 Collision detection approaches

The main problem which needs to be solved in a case of the each physical phenomenon simulation is some kind of collision detection, i.e. decide if two objects collide. In our case, one object is always the avatar and the other one is an arbitrary model from the virtual world. Thus, we shall investigate possible approaches for resolving collision detection.

There are existing collision detection solutions available in JavaScript, but most of them are targeted at 2D. Solutions available for 3D such as oimo.js[1]) or ammo.js[2]) are more of a whole physics engines, very robust. Because collision detection necessary for our application can be simple – we just need to handle *avatar vs everything* collisions, and there are no moving objects – using such solution would cause more harm than good (performance overhead, potentially complex integration with the application), which is why we have decided to implement simple collision system by ourselves.

The first thing is to decide on the type of a collision detection which will be implemented. There are two main types of a collision detection:

- Continuous
  - based on a prediction of object trajectories
  - objects never intersect "physically"
    - it is known exactly if they are about to collide and the moment of collision, thus their positions are updated accordingly so they will not collide with each other
- Discrete
  - advances the physical simulation by a small amount of time and then checks if any of objects collide
  - in contrast to the continuous collision detection, objects do collide "physically" during this method
    - the moment of collision is not known (it is "missed")
    - when the collision is detected, positions of objects are updated accordingly so they will not collide with each other **anymore**

The main advantage of the continuous collision detection is its precision – we never miss the exact moment of a collision, but the obvious disadvantage is the performance along with a code complexity, since various non-trivial computational geometry methods have to be used. Because of that, **discrete collision detection was chosen** (the simulation is advanced by elapsed time between the two consecutive frames, i.e. we call the collision detection algorithm each frame). This can introduce **the problem of**

[1]) `https://github.com/lo-th/Oimo.js/`

[2]) `https://github.com/kripken/ammo.js`

potentially missed collisions. When objects move too fast, a collision may actually occur between (two consecutive) time steps, which means we will not know about it; this is commonly referred to as the "*bullet through paper*" problem (see Figure 8.1). Fortunately, objects (papers) in our application are fairly large, so missed intersections with the avatar (bullet) should not be an issue.

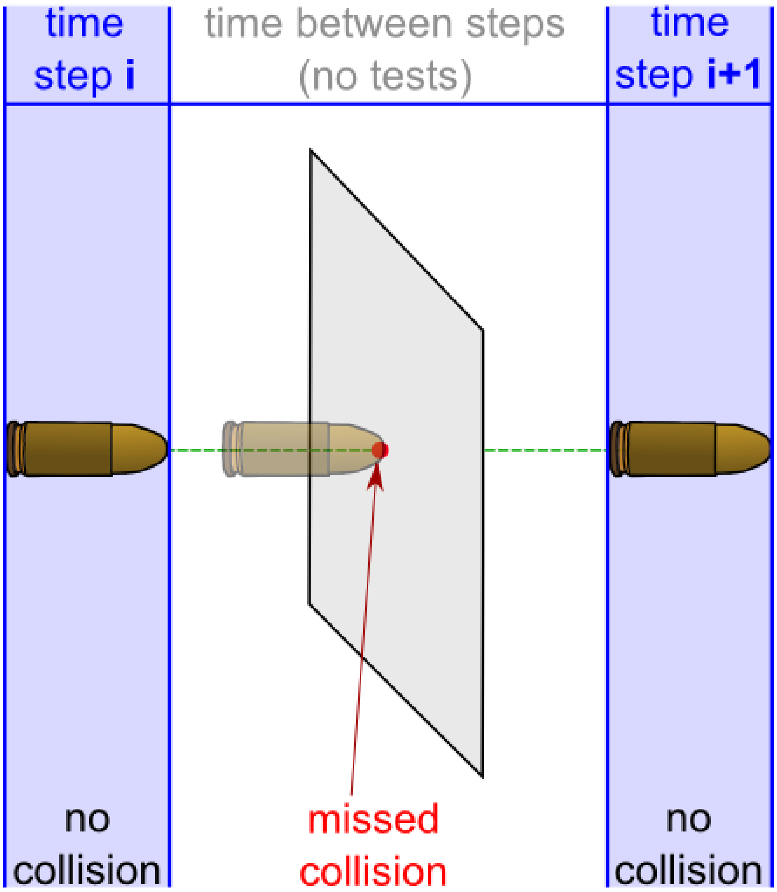


Figure 8.1. Missed collision – bullet through paper problem

We will consider two of the common collision detection solution types:

- object-based
  - works with geometry of objects
  - transforms the collision detection into a computational geometry problem
    - e.g. *point vs triangle intersection*, *point vs sphere intersection*, ...
- heightmap-based
  - popular solution for representing a natural terrain, but works with urban areas as well
  - 2D matrix of values that determines height of the terrain (Y) for each location at the ground plane (XZ plane in WebGL)
  - collision detection carried out by a simple comparison of the avatar's Y position against the heightmap (terrain Y)

Since the avatar has a body (although invisible), we should not represent it just as a point. Instead, we should use some type of a bounding volume (the most convenient being an Axis-Aligned Bounding Box – it provides a good approximation of a human figure for purposes of the collision detection and is simpler to work with than e.g. a more fitting cylinder) or at least a line segment. Virtual world models consist of triangles, therefore the computational geometry problem which needs to be solved in the **object-based approach** is *AABB/line segment vs triangle intersection* problem. Be that as it may, it is not sufficient just to implement an intersection algorithm – if the AABB/line segment (avatar) is colliding with a triangle (object boundary), we also need to know how to update the position of the AABB/line segment so it will not be colliding with the triangle anymore. For this, it is also necessary to compute the *"least translation vector"*, representing the smallest amount of translation which needs to be applied to the AABB/line segment position in order to stop it from colliding with the triangle. Example is shown on Figure 8.2. The "platform" consisting of several triangles is penetrated from above by the line segment AB. The least translation vector is part of the line segment AB between the point A and the collision point C, directed towards the point B. If we translate point A of the line segment AB by this vector, the line segment will no longer collide with the platform and will "sit" on the penetrated triangle. Also, we cannot forgot to mention that it is an absolute necessity to employ some space-partitioning data structure to reduce number of triangles which will be tested for a collision.

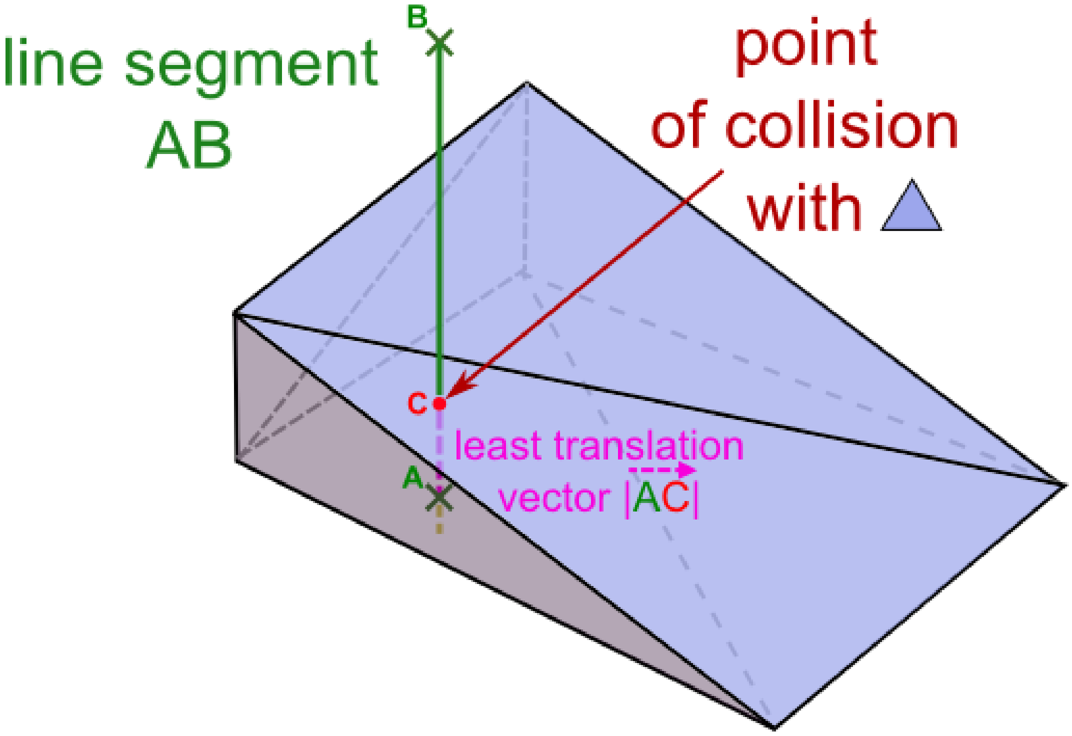


Figure 8.2. Line segment vs triangle intersection – the least translation vector

The other approach based on the **heightmap** requires a small pre-processing step (rendering the scene using an orthographic projection from above), but no space-partitioning data structure is needed. The main principle behind the collision detection is that we find out which coordinates in the heightmap correspond to the current avatar's 3D position, look up the height value at such coordinates and assign it as the new Y position for the avatar. This takes care of walking around a terrain. Collisions with objects can be detected as well, because coordinates where e.g. buildings are will have much larger height value than the ground, so when there is a sudden change in height, we know that a collision has occurred.

Given the fact that we have to perform extremely similar pre-processing step for the purpose of the current avatar's area detection and map displaying (as was discussed in Chapter 7), less complexity and certainly a faster performance, **we have decided to go with the heightmap-based approach**. However, there is one particular disadvantage of heightmaps which causes us trouble: not being able to handle "overhangs". Passageways between the castle courtyards are an example of such overhang cases – there are several possible height values for a single XZ coordinate. Example of this situation is depicted on Figure 8.3, where a building containing a passageway is shown from the two different views, with points A and B having the same XZ coordinates, but different Y coordinate. This situation is unpleasant, but we will describe how to deal with it.

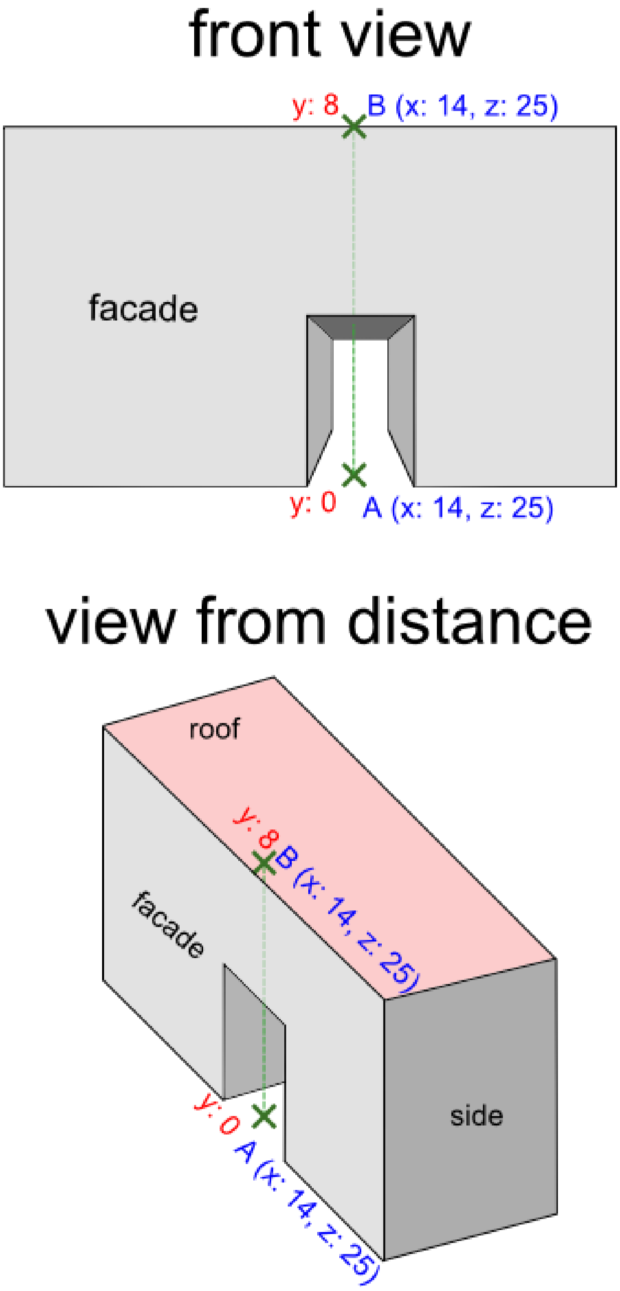


Figure 8.3. Example of an overhang – passageway

8.3 Heightmap implementation

Heightmap for the application is represented in the standard way – by a raster image (texture). The area captured in the heightmap corresponds to the 600×600 m area in the real world – the **whole image** on Figure 8.4 shows the **exact captured rectangular area** from the aerial view, with the Prague Castle outlined in red. For the sake of completeness, area which is accessible in the final application is colored in blue. The resolution of the heightmap was chosen to be 1280×1280px, yielding a precision of roughly about 0,5 m per pixel[1]), which is sufficient enough.

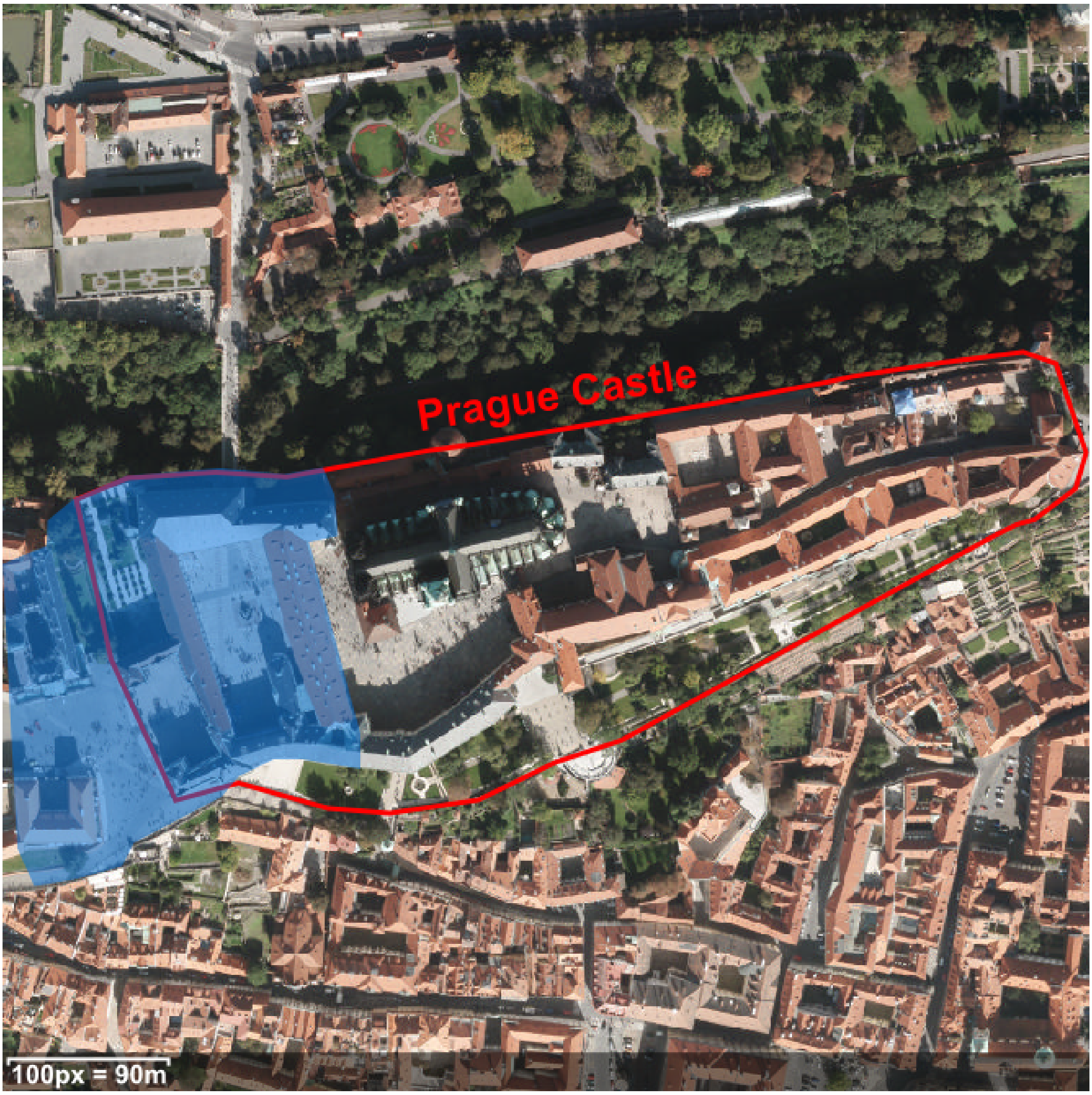


Figure 8.4. Area captured by the heightmap – aerial view [50]

[1]) 1px in the heightmap represents 0,5 m in the virtual world (and reality as well)

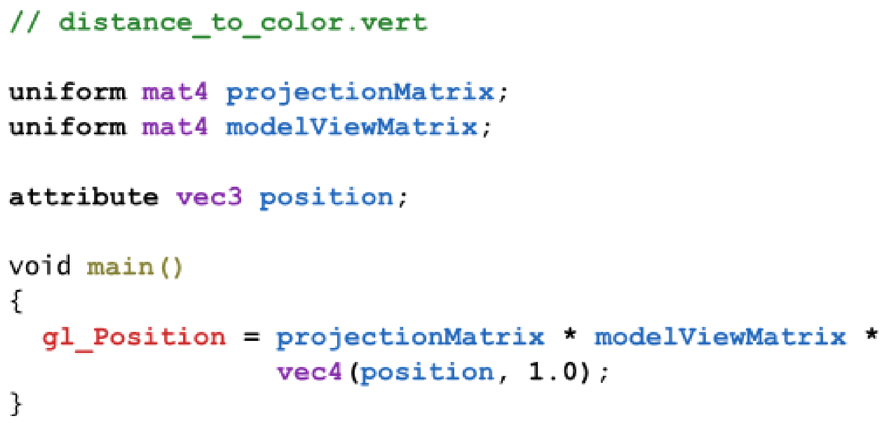
8.3.1 Creation of the heightmap

Creation of the heightmap is a part of the pre-processing step. We use an orthographic projection and set the camera parameters so that the area of the Prague Castle meant for exploring is fully visible:

```
// orthographic projection parameters
left, right, bottom, top = 300
near = 0.1
far = 300

// camera parameters
position = vec3 (60, 150, 10)
rotation = vec3 (-PI/2, 0, 0)
```

Note that in our application, 1 WebGL unit equals 1 m in the reality. The walking area of the Prague Castle (i.e. ground) is for the most part flat – Hradčany Square, all Courtyards, generally everything up to St. George's Square have roughly the same altitude; we have chosen this altitude to be 0 m in the application (i.e. meshes representing the ground have their Y position around 0). We set all clipping planes to 300, which makes desired area of the Prague Castle visible. Far plane is set to 300, which means that we are able to capture vertical range of 300 m into the heightmap. Because the camera is positioned at 150 m above the ground (`position.y`), it means we can capture vertical range of 150 m above and 150 m below the Prague Castle ground. Values of `position.x` and `position.z` are most easily found by a short session of trial-and-error. Camera `rotation` contains the standard setup for the top-down view.

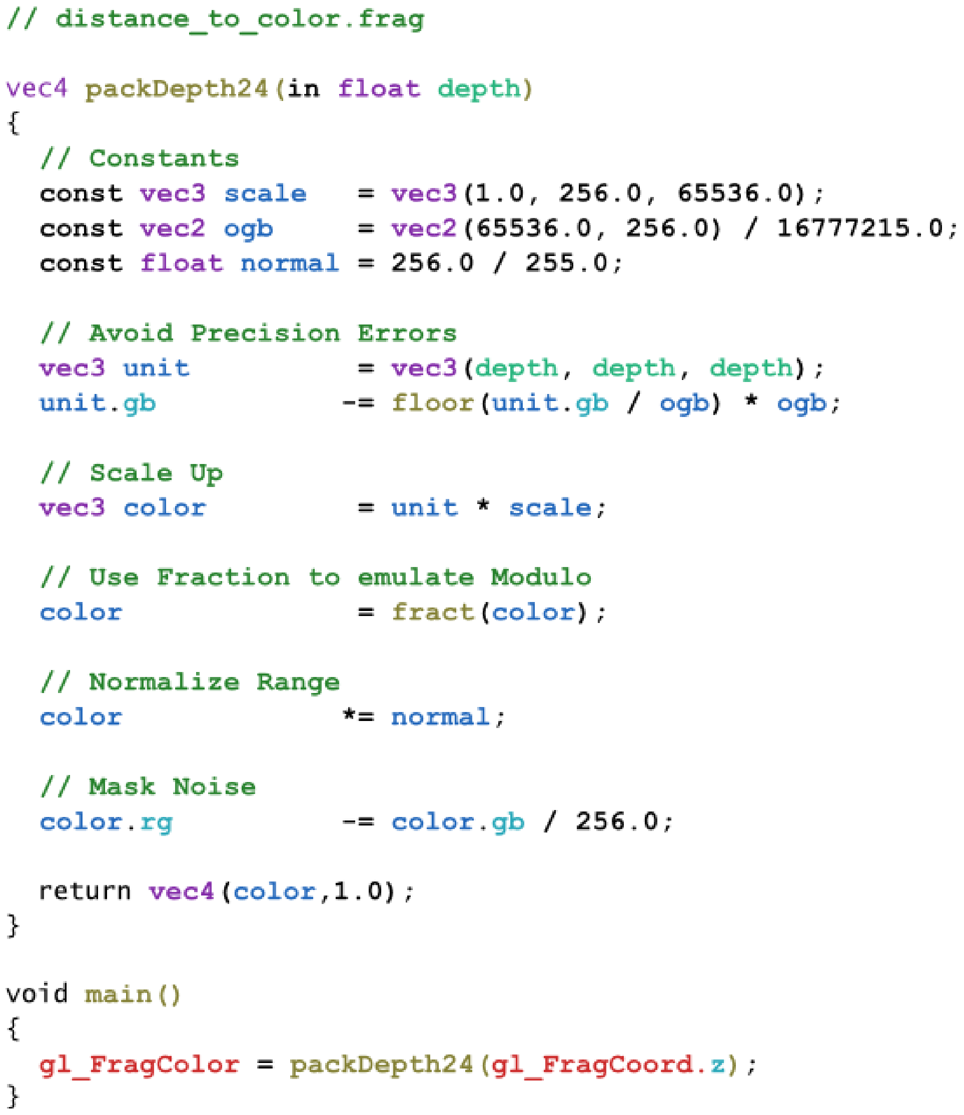
The resulting projection-view matrix of the camera is saved. Then, all collidable objects such as buildings, ground etc. are **rendered with a special WebGL program which encodes distance of a fragment from the camera into its color**[1]) (also referred to as *"packing depth into a texture"*). The vertex shader of the program can be found on Figure 8.5 and does not do anything special – it just contains the classic line which produces clip coordinates of a vertex. Fragment shader (Figure 8.6) is a little bit more interesting – it takes depth of a fragment in window coordinates (i.e. distance from the camera, ranging from 0 to 1) and produces a corresponding color by calling the `packDepth24` function from [51], which uses RGB portion for the encoding, thus we have a precision of 24b available.



```
// distance_to_color.vert

uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;

attribute vec3 position;

void main()
{
  gl_Position = projectionMatrix * modelViewMatrix *
                vec4(position, 1.0);
}
```

Figure 8.5. Encoding depth to a color – vertex shader

[1]) This program is also used for picking (as was discussed in 7.3.6)



```
// distance_to_color.frag

vec4 packDepth24(in float depth)
{
  // Constants
  const vec3 scale   = vec3(1.0, 256.0, 65536.0);
  const vec2 ogb     = vec2(65536.0, 256.0) / 16777215.0;
  const float normal = 256.0 / 255.0;

  // Avoid Precision Errors
  vec3 unit           = vec3(depth, depth, depth);
  unit.gb            -= floor(unit.gb / ogb) * ogb;

  // Scale Up
  vec3 color          = unit * scale;

  // Use Fraction to emulate Modulo
  color               = fract(color);

  // Normalize Range
  color              *= normal;

  // Mask Noise
  color.rg           -= color.gb / 256.0;

  return vec4(color,1.0);
}

void main()
{
  gl_FragColor = packDepth24(gl_FragCoord.z);
}
```

Figure 8.6. Encoding depth to a color – fragment shader

To save the result of the pre-processing, we can retrieve what is rendered by using method of the `<canvas>` tag called `canvas.toDataURL()`, which produces base64[1]) encoded PNG file with the rendered image that can be saved on the disk.

8.3.2 Using the heightmap

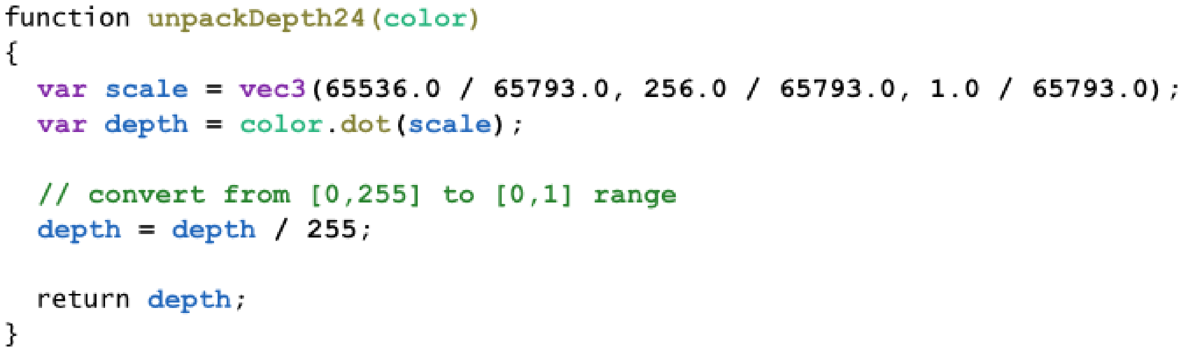
At the beginning of the application, we load the image with the heightmap and save its pixel data to a 1D array in the memory. To obtain height value from the heightmap, the first thing to do is to convert the avatar's 3D position in the world into the corresponding 2D position in the heightmap. Using the view-projection matrix which was saved during the heightmap creation, it is possible to transform current 3D coordinates of the avatar to the corresponding 2D coordinates in the heightmap similarly to as when transforming vertices from a 3D world onto the screen:

- Avatar's 3D coordinates are transformed by the saved view-projection matrix into the clip coordinates

[1]) https://en.wikipedia.org/wiki/Base64

- Perspective division is not needed
  - that is because orthographic projection was used
  - normalized device coordinates are thus same as clip coordinates
- Normalized device coordinates are mapped onto the heightmap image (viewport transformation)

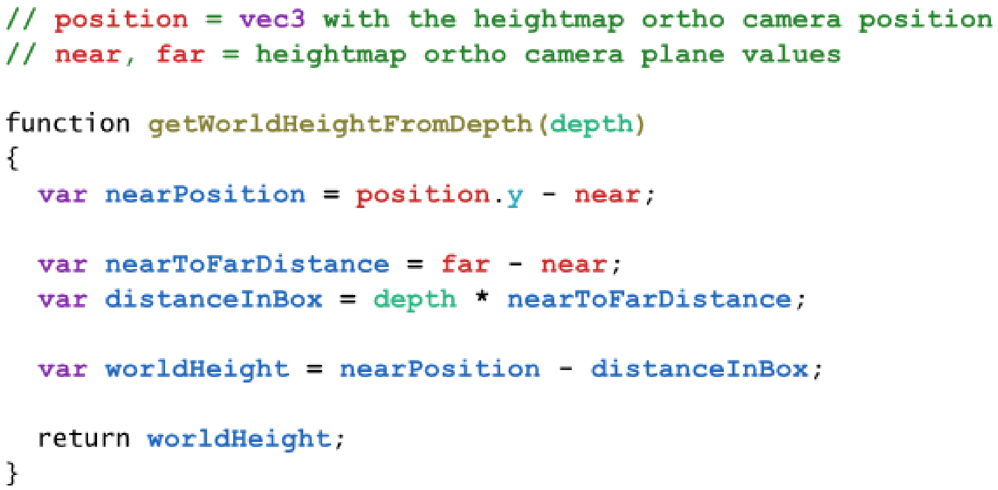
A color of the pixel at the obtained 2D position is retrieved and decoded into a number value using an inverse function `unpackDepth24` (JavaScript) to the `packDepth24` (GLSL) from Figure 8.6. The JavaScript code for the decoding function (again from [51]) can be seen on Figure 8.7



```
function unpackDepth24(color)
{
  var scale = vec3(65536.0 / 65793.0, 256.0 / 65793.0, 1.0 / 65793.0);
  var depth = color.dot(scale);

  // convert from [0,255] to [0,1] range
  depth = depth / 255;

  return depth;
}
```

Figure 8.7. Decoding depth from a color – JavaScript

As was explained in the previous section 8.3.1, the decoded number represents the depth (“distance”) of the pixel from the camera that was used during the process of the heightmap creation. The depth is expressed as a number in the range from 0 to 1, where value of 0 means that the pixel was right at the camera’s near plane, 0.5 means that it was exactly halfway between the near and far plane, 1.0 means it was at the far plane etc. What we would like to know is: what world height (i.e. Y coordinate in the world coordinates) corresponds to the given depth value? The formula is straightforward and can be seen on Figure 8.8. By computing the camera’s `nearToFarDistance`, we can convert `depth` value to meters (`distanceInBox`). The only catch is that to obtain the correct world height, we must subtract the `depth` in meters not from the camera’s `position`, but from the position of its `near` plane (`nearPosition`).



```
// position = vec3 with the heightmap ortho camera position
// near, far = heightmap ortho camera plane values

function getWorldHeightFromDepth(depth)
{
  var nearPosition = position.y - near;

  var nearToFarDistance = far - near;
  var distanceInBox = depth * nearToFarDistance;

  var worldHeight = nearPosition - distanceInBox;

  return worldHeight;
}
```

Figure 8.8. Computing world height from a depth

8.3.3 Heightmap layers

The Prague Castle contains several passageways, mostly between the courtyards. This poses a problem for a heightmap, being an “overhang” scenario, with more than one height (Y) defined at single position (X,Z). Example was shown on Figure 8.3. We will now discuss how to cope with this shortcoming.

Up until now, we have assumed the standard case when one pixel in a heightmap is used for encoding one height value into its color. In such case, a pixel can utilize for its depth encoding up to four RGBA channels (32b). One practical solution, assuming that each height value require same amount of space, may be storing height values either in separate color channels (8b per value, we could store four such values) or in a pair of channels (16b per value, we could store two such values). Anyway, 8b are not enough: even if we reduce our chosen vertical range from 300 m to the lowest possible range which is about 100 m (the difference between the lowest and the highest point in our virtual world, i.e. between the ground and the main tower of St. Vitus Cathedral), we would not be able to store a height difference smaller than $\frac{100}{2^8}$ m, which is about 39 cm. That would be a problem, because there are smaller objects than this appearing in our virtual world – for example stairs are usually about 10 to 15 cm, so it would cause problems with walking on them properly. 16b per one height value are sufficient, but we can only have two such values stored in a pixel. The problem is that we need to store three values in each pixel for passageways to work with a heightmap:

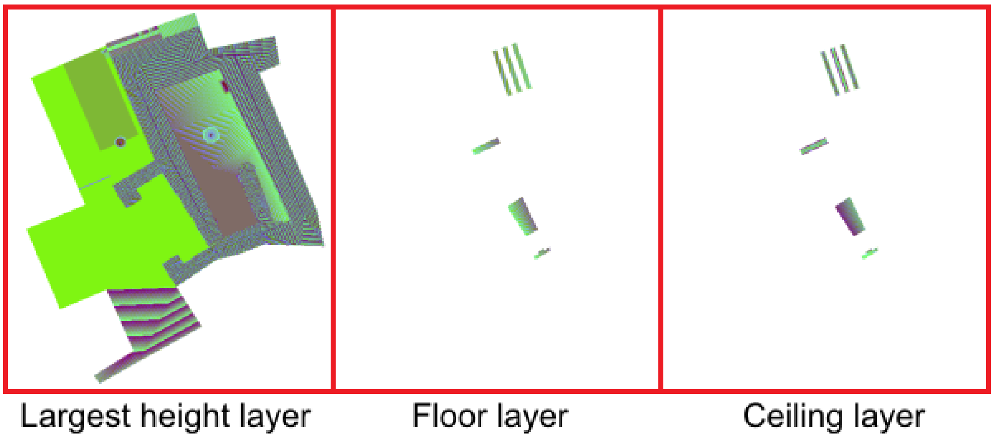
- Largest height
  - the largest Y position of all triangles which project to the pixel
  - typically height of the ground, roofs, objects (fountain, ...)
  - always defined
- Passageway area – ceiling height
  - Y position of a passageway ceiling triangle, which is at the position that projects to the pixel
  - undefined if the pixel is not in a passageway
- Passageway area – floor height
  - Y position of a passageway floor triangle, which is at the position that projects to the pixel
  - undefined if the pixel is not in a passageway

Since we cannot store all three values comfortably into one heightmap, it was decided that **three separate heightmaps (layers) will be used**, each storing one type of the values mentioned above. This means triple memory requirements for the entire heightmap, but that is an acceptable price to pay concerning the fact that the heightmap images do not need to be on the GPU, but rather in the main memory. In the proposed system of three layers, it is possible to move inside passageways the same way as everywhere else (i.e. by walking, flying, ...), although the system cannot handle multiple passageways above themselves. Fortunately, there are no constructions like that in the Prague Castle area, so it is not a problem.

Generating the layers is almost the same as generating the heightmap as was described in 8.3.1. The only thing different in the generation of each layer is which models are rendered:

- Largest Height layer
  - all models are rendered
  - (the exact same process described in 8.3.1)
- Passageway area – ceiling height layer
  - only triangles which are part of some passageway's ceiling are rendered
    - such triangles must be marked in the SketchUp (explained below)
    - we have to render both front and back faces of ceiling triangles – otherwise a ceiling would not be visible either by the user (if the direction of the front face is positive Y) or the orthographic camera which is generating the heightmap (if the direction is negative Y)
- Passageway area – floor height layer
  - only triangles which are part of some passageway floor and possibly any other models standing inside them, e.g. small statues, are rendered

Triangles that belong to a floor or a ceiling must be manually marked in SketchUp. This is done by selecting all triangles which are part of a floor/ceiling, creating a new group from them and assigning it a name which starts with a prefix "`floor_`" or "`ceiling_`". How the different layers of the heightmap look like is shown on Figure 8.9.


Largest height layer

Floor layer

Ceiling layer

Figure 8.9. Three layers of the heightmap

8.4 Principles of the application's collision detection

Let us take a look at our simple collision system from a higher level. Each time when the avatar's position changes, the collision system checks if a collision has occurred. If no collision has occurred, the avatar's position can safely change and we are done. If the collision does occur, the solution seems obvious at the first glance: reject the new position and keep the old one. However, this could lead to the situation when the avatar will stop its movement too far from an obstacle (e.g. wall) although it is clearly still possible to move closer. It happens when an increase in the avatar's position during the advancement of the simulation is bigger than the avatar-to-obstacle distance. It is related to the bullet through paper problem which has been mentioned earlier (see Figure 8.1), in which the collision is missed. In this situation, **the collision is not missed, it is detected like it should be, but that prevents the avatar from a further movement in its direction**. What would we like to know is the *least translation vector* which we have talked about as well. The bad news are that computation of this vector is far harder than in the case of *AABB/line segment vs triangle intersection*, because we work with a different kind of information (image instead of a geometry).

One way of finding the least translation vector involves ray-casting in the heightmap. To be precise, we are searching for the "opposite" of the least translation vector – **the "largest" translation vector**, i.e. the largest vector by which we can translate the avatar's position without the avatar being in the collision. We know that there is a sharp transition in the heightmap between places that have much different heights, which allows us to check for collisions not just with the terrain, but also with walls and such. By casting a ray from the avatar's position to the rejected (colliding) position and inspecting height values of intersected pixels along the ray, we can find the boundary (= two pixels along the ray which heights differ too much) between e.g. the ground and a wall, which is all we need for figuring out the largest translation vector. The problem is that this position is in the 2D heightmap image space, but we need it in the 3D object space. We can unproject this 2D position to obtain (X,Z) coordinates in 3D, however, we are limited by the precision of the heightmap. To put it simply, there are several 3D positions which project into the given 2D position, but the given 2D position unprojects just into one 3D position. While our heightmap precision of 0,5 m is enough for the general collision detection, it is not enough for figuring out the largest translation vector, because being possibly off by 0,5 m does not solve our problem of the premature stopping in front of obstacles at all.

We could increase the resolution of the heightmap to increase its precision, but there is a simpler (and more memory-effective) solution which approximates the largest translation vector: each time when a collision occurs, we **divide the avatar's position increment into smaller parts, and each time when the smaller increment is applied, we try to detect the collision again**. A specified number of these additional collision checks are performed (we call this number the *"collision budget"*). The first additional check moves the avatar with the amount equal to $\frac{1}{2^1}$ of the avatar's position increment, the second additional check moves the avatar even further with the amount equal to $\frac{1}{2^2}$ of the avatar's position increment and so on. Formally, this can be written as:

$$checked\ position(n) = orig.\ avatar's\ position + position\ increment * \frac{1}{2^n}$$

$$\text{where}$$

$$n = \text{number of an additional collision check}$$

The additional collision checking is terminated when the currently checked position is in the collision. The largest translation vector approximation is then obtained as the vector between the original avatar's position and previous additionally checked position (that is the last one which has not been in any collision). Illustration of the additional collision checking is shown on Figure 8.10. The collision budget of three additional checks proved to be sufficient. The reason why we do not have the "infinite" collision

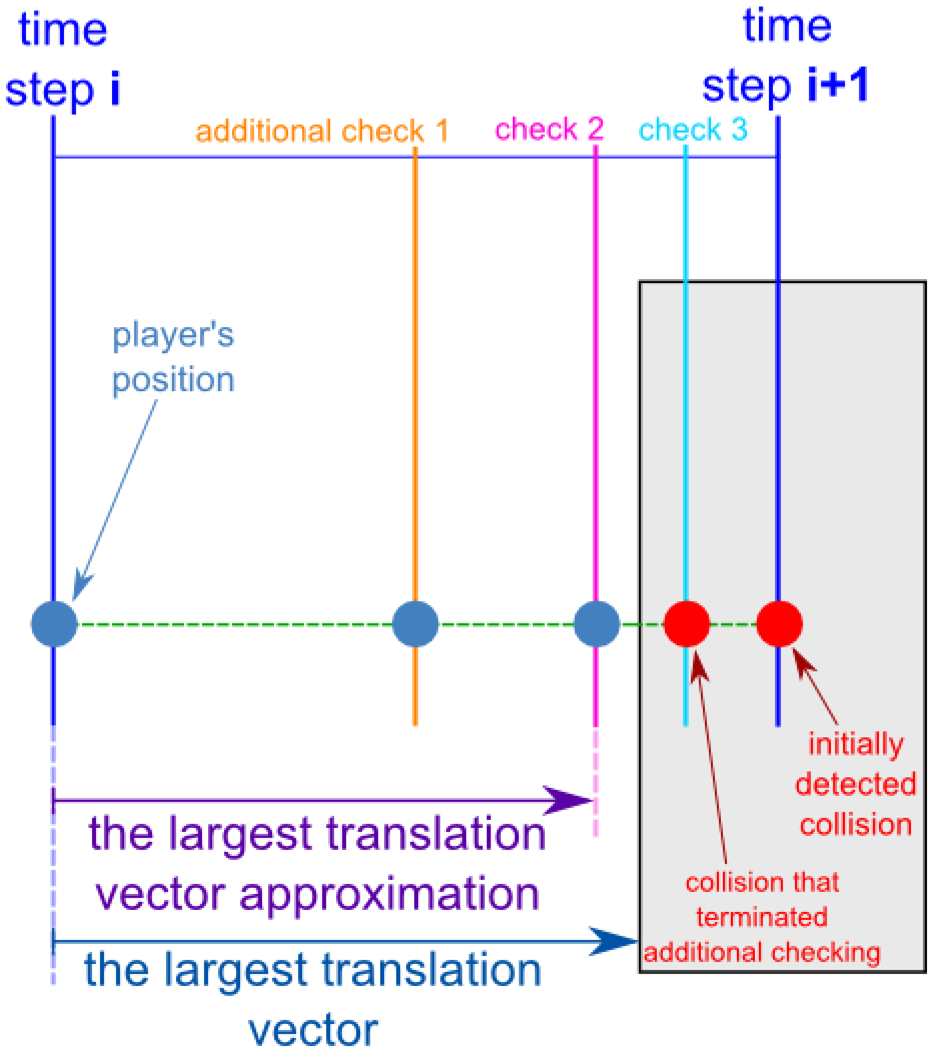

time
step i
time
step i+1
additional check 1
check 2
check 3
player's position
initially detected collision
collision that terminated additional checking
the largest translation vector approximation
the largest translation vector

Figure 8.10. Additional collison checking – finding the largest translation vector approximation

budget is that there are special cases where too much of additional checks may be performed – happens when the initially detected collision is right beyond the obstacle.

It is also important to mention that the value of the near camera plane may have to be adjusted. If its value is too big, even though the collision detection is working like it should, it will appear as if avatar is in the obstacle, colliding, since the obstacle's texture will not be visible and we will see through.

8.5 Implementation of collision detection

This section will cover implementation details of our simple collision detection system. First, let us divide the avatar's movement into the two main types:

- Horizontal
  - moving along the terrain (walking)
- Vertical
  - moving along the Y direction (falling or flying)

Because our application simulates gravitation, these movement often happen simultaneously. Let's now take a look at which situations do collisions happen exactly. There are several events related to them which take place in the application:

- Walking on an uneven terrain
- Walking into a wall
- Falling
- Flying
- Walking through a passageway

We will now examine these events, what is similar for them and how are they handled from the standpoint of our collision system.

Walking on an uneven terrain, walking into the wall: this is the most usual case, where the avatar walks on the terrain, just like we do in the real world. The user performs only horizontal movements with the avatar, i.e. only X and Z coordinates can be changed by the user's actions. The Y coordinate is always computed based on the X and Z coordinates by a lookup into the heightmap. This simulates an effect of walking on an uneven terrain, since the avatar is always at the highest point for its current world position.

In any case, to correctly simulate the reality, we have to make sure that the avatar cannot walk on a terrain which is too steep. The solution to this is the same as the solution which prevents the avatar from walking through the walls – height at the current avatar's position and at his potentially new position is compared:

```
heightDiff = newHeightmapHeight - currentHeightmapHeight
```

If the value of `heightDiff` is greater than ε (we use $\varepsilon = 1{,}0\,\mathrm{m}$), it means the new height is a lot bigger than the old height, so we can conclude that avatar cannot access the new position. Note that it is not necessary to differentiate whether the avatar cannot access the new position because the terrain is too steep or because there is a wall (or other obstacle) in front of him, since from the collision system's standpoint, steep terrain and wall (obstacle) are one and the same thing. We shall point out that the terrain in the context of our collision system is not just the ground, it can be also roof or similar object – strictly speaking, it is simply the highest point at a certain heightmap position.

Avatar falling: the usual scenario is that the avatar walks on the roof and then "jumps" off it. Since our application simulates gravity, naturally, the avatar starts to fall to the ground and after a certain amount of time, it will land.

Before we explain how we detect that the avatar should fall, let us introduce avatar's "**air status**", which tells us whether the avatar is in the air. Avatar's air status can have one of these two (three) values:

- not in air
  - walking on a terrain (ground, roof, etc.)
- in air
  - from implementation reasons (will be discussed), we further differentiate between the two "in air" states:
    - in air for a long time
    - just got into air

Finding out value of the avatar's air status is carried out by comparing avatar's Y position and value from the heightmap at the avatar's (X,Z) position:

```
heightDiffAir = avatarY - currentHeightmapHeight
```

This equation is similar to when we were checking collisions with the terrain and walls. If the difference (`heightDiffAir`) is greater than ε (this time, we use a slightly smaller $\varepsilon = 0{,}8\,\mathrm{m}$), the avatar's Y position is much bigger than the current heightmap height, meaning that the avatar is far above the terrain and thus `air status = in air` (otherwise, if the difference is smaller than ε, it is not in the air and thus `air status = not in air`).

Now, when the `air status = in air`, we need to differentiate between two situations: whether the avatar just got into the air (e.g. user has "jumped" off a roof) or whether the avatar is already falling, being in the air for a long time. Particularly important is the former situation, since it is an instant when the avatar starts to fall. When this situation happens, current avatar's Y position and current time (obtained e.g. by JavaScript function `Date.now()`) are saved:

```
avatarY_fallStart = avatarY
fallStartTime = Date.now()
```

The actual falling happens in the case of `air status = in air for a long time`. The new Y position of the avatar, valid for the current frame, is computed by using the previously saved `avatarY_fallStart`, and `fallAmountY` values:

```
newAvatarY = avatarY_fallStart - fallAmountY
```

The `fallAmountY` is computed based on the previously saved `fallStartTime`. We consider uniform gravitational field without air resistance in our application and freefall as a vertical motion of an object (avatar) falling a small distance close to the surface of a planet (ground), which is a good approximation for our purposes. The equation used for computing `newAvatarY` during the fall is following [52]:

$$y(t) = -\frac{1}{2}gt^2 + v_0 t + y_0$$

where:

v_0 is the initial velocity $(\frac{m}{s})$

y_0 is the initial altitude (m)

$y(t)$ is the altitude with respect to time (m)

t is time elapsed (s)

g is the acceleration due to gravity (9.81 $\frac{m}{s^2}$ near the surface of Earth)

Lets modify the equation so it matches our implementation. First, we rename the variables:

$$y(t) = \texttt{newAvatarY}$$

$$t = \texttt{Date.now()} - \texttt{fallStartTime}$$

$$y_0 = \texttt{avatarY\_fallStart}$$

Next, we can leave out the second term, since the avatar does not have any initial velocity. Also, we will rearrange the terms, putting y_0 at the start. This leaves us with the modified equation used in the implementation:

```
newAvatarY = avatarY_fallStart - 0.5*g*(Date.now() - fallStartTime)^2
```

The end of the avatar's falling is given by the transition between the avatar's `air state = in air` to `air state = not in air`, which happens when the avatar's Y position is below the terrain's Y position (heightmap height). The nice thing is that the avatar will be placed correctly on the terrain (with no additional collision checks needed, in contrast to the case of walking and walls) right away, because when the `air state` becomes `air state = not in air`, avatar's Y position will automatically obtain the same value as terrain's Y position.

Earlier, we have discussed the events of walking (which is a horizontal type of movement). The walking and falling (vertical movement) can occur at the same time, e.g. user can move the avatar wherever he likes during falling. Collisions against obstacles will still be working like in the usual case when walking on the terrain.

In the **Avatar flying** event, the only difference compared to the walking is that avatar can move not only horizontally, but vertically as well. Gravitation is turned off, so consequently, avatar's Y position is fully determined only by the user's movements. In other words, the user can move the avatar freely in all directions.

We intentionally mention the event of **Walking through passageways** as the last one, because there is nothing very special to collisions inside passageways. What needs to be done is the test if the avatar is in a passageway or not. This test is performed at the very start of each collision detection call. After we obtain the avatar's 2D position in the heightmap, we look to the *Passageway area floor height layer*. If the pixel at the position is undefined, we are not in a passageway, otherwise we are – in that case, we start to use *Passage area floor height layer* for computation of the avatar's Y position during the collision events instead of the normally used *Largest Height layer*. Also, the movement is limited from above by a passageway ceiling height (normally, it is not limited).

Table 8.1 contains important collision events' properties summary, where for each situation, we can see in which directions can avatar move and based on which part of the collision detection system is its Y position calculated.

Action	Directions user can control movement in	Avatar's Y calculcation
Walking	X, Z	heightmap
Falling	X, Z	freefall simulation
Flying	X,Y,Z	none

Table 8.1. Collision events' properties summary

Chapter 9 Performance optimizations

Despite the fact that we have a quite powerful machines available today, even simpler applications with 3D virtual worlds can run horribly slow. Similarly to other computer science related problems, it is not sufficient to tackle the rendering process using a naive approach. It is crucial to utilize effective approaches and apply optimized solutions to get most out of the available hardware. Our application is no exception and running it at smooth speed without efforts put into performance optimizations would be impossible. This chapter will explore shortcomings of the naive approach to rendering (put into the context of Three.js, but the principles apply generally) and explain how it can be improved using the technique of "**Mesh merging**" along with the implementation details. Usage of popular rendering optimization techniques such as LOD, texture atlases, frustum and occlusion culling is discussed as well.

9.1 Goals and considerations

Our goal, according to the specified project requirements, is to achieve 30+ FPS on the supported platforms. Moreover, even though the final application does not contain the whole area of the Prague Castle, we shall design optimizations with it in mind to allow for adding the remaining places without the fear of making the application slower due to such addition. As for the implementation, we shall strive for approaches that do not modify inner parts of Three.js. The discussed optimizations focus only on rendering speed and reflect the virtual walk nature of our application – mainly the fact that it has mostly static geometry and contains no animations of humans or animals.

9.2 Three.js and meshes

Before we start the actual discussion about optimizations, it is necessary to explain how Three.js operates in regards to the rendering. Note that in the following text, we assume that all faces are triangles. Also some class names etc. were altered and behavior processes of Three.js were simplified a little so we can better focus on the principles.

The most important class from Three.js in this matter is `Mesh` class, which holds information about a 3D model – mainly its matrix transformations, used materials and geometry. It is possible to classify meshes into the two categories, depending on the number of materials they use:

- One-material mesh
  - uses only one material
  - all its triangles must use this material

- Multi-material mesh
  - uses multiple materials
  - triangles can have different materials

Geometry data of a `Mesh` is available in a form of an object. How the geometry information is stored in the memory and used is determined by a Three.js class which was used to instantiate such object. There are two such classes (types):

- `Geometry`
  - standard data structure for storing a geometry information
  - holds all data necessary to describe a 3D model
  - possible to easily access individual data items
    - arbitrary vertex positions, texture coordinates, faces definitions, ...
    - e.g. position of the 2nd vertex of the 27-th triangle of a model
  - data items are provided in an easy-to-work-with format and manner
    - e.g. we can retrieve position as `Vector3`
- `BufferGeometry`
  - an efficient alternative to `Geometry`, but harder to work with
  - stores all data within buffer(s)
    - reduces cost of passing data to the GPU
  - rather than accessing position data as `Vector3` objects, color data as `Color` objects etc., we have to access raw data from the appropriate attribute buffer
  - faces definitions in `Geometry` are translated to `DrawingCall` objects
  - draws geometry directly from the buffer(s)

The types of geometries can be converted between each other. Three.js internally uses `BufferGeometry` for rendering, so even if the application code explicitly uses only `Geometry`, the `BufferGeometry` is created nevertheless (although it is not an optimal conversion yielding the highest speed). So the difference is that with `Geometry`, data is stored in a more "user-friendly" way, it occupies more memory and it is not really possible for us to speed up the rendering process. With `BufferGeometry`, it is exactly the opposite. A simplified graphical overview of the explained hierarchy is depicted on Figure 9.1. Especially important classes are `Face`, which instances fully define one face (triangle), and `DrawingCall` class, which is a definition of one WebGL drawing call which will be performed during the rendering process. The `Attribute` class is a container holding information about mapping of per-vertex attributes (position, normal, ...) and is resembling arguments of the classical OpenGL/WebGL method `glVertexAttribPointer`[1]).

It is also worth to mention that we use a non-indexed geometry (i.e. `glDrawArrays()` is used to render the data). Three.js allows a developer to render an indexed geometry (rendering by `glDrawElements()`), but relevant available loaders (Collada loaders in our case) do not create an indexed geometry. We would have to deal with it entirely on our own. Implementation of the custom loading process, preparing such geometry and working with it would bear a significant added complexity, thus we have settled with a non-indexed geometry.

[1]) https://www.khronos.org/opengles/sdk/docs/man/xhtml/glVertexAttribPointer.xml

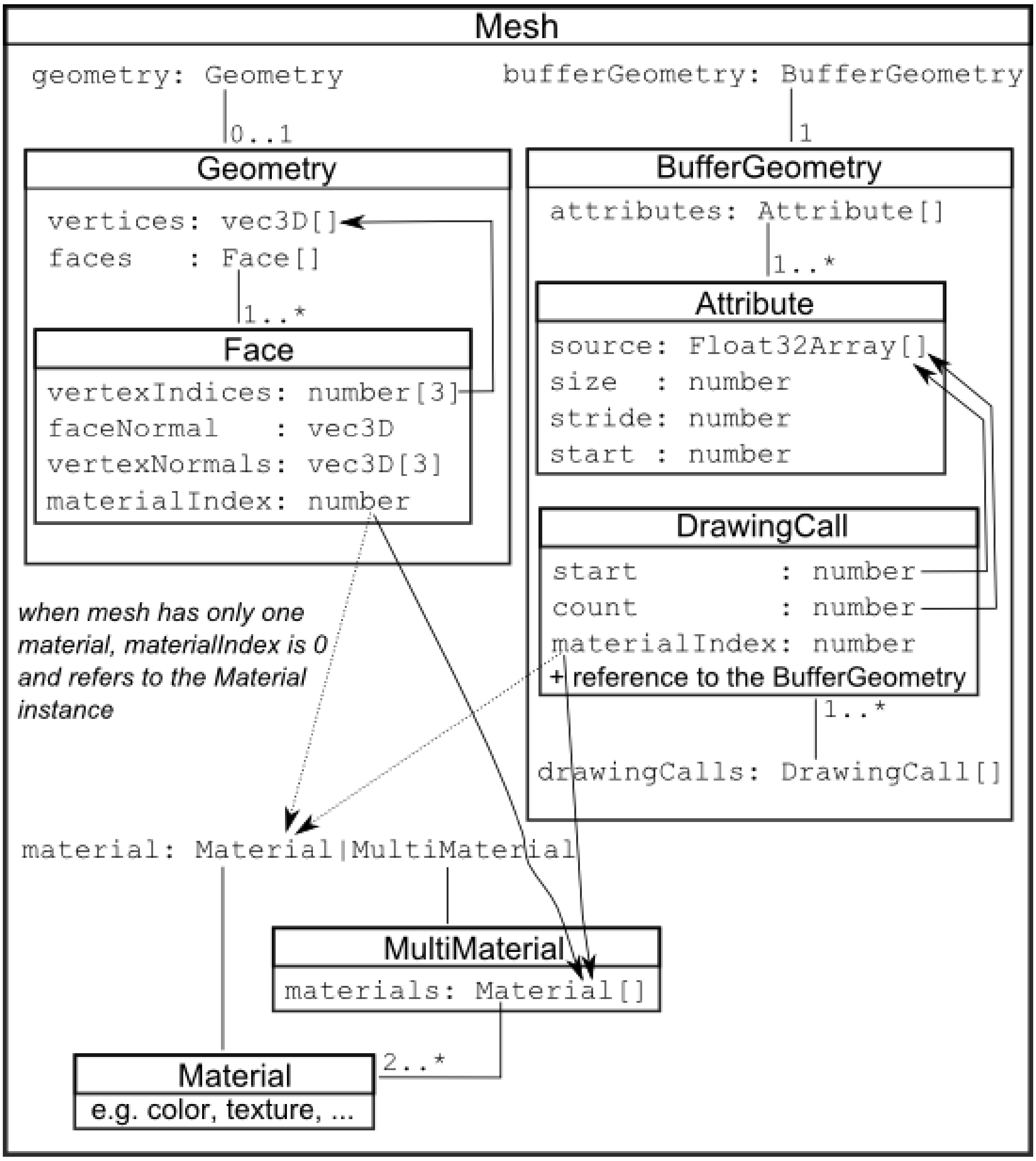


Figure 9.1. Three.js – structure of `Mesh` class

9.3 Rendering approaches

When a scene is loaded via Three.js "naively", i.e. just by calling appropriate loader for used 3D models, the performance can be downright terrible even for not very complex scenes. In order to understand why the framerate is being so low and overcome this problem, it is necessary to analyze the situation and see what Three.js is doing under the hood.

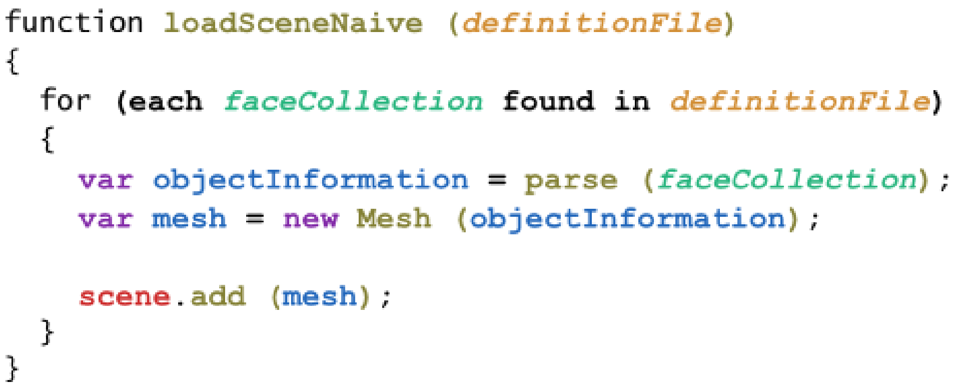
9.3.1 Naive rendering approach

The testing scene which we have used for exploring different loading approaches is actually an unfinished, development version of the scene appearing in the final application. The content of the unfinished scene was copied multiply times and these copies were inserted alongside each other, forming the entire testing scene. The testing scene consisted of:

- 304 000 triangles
- 140 materials (textures)
- one ambient and directional light source
- shadows via shadow mapping
  - other than that, no visual effects

Lets assume that the whole scene is defined in some kind of a scene definition file (e.g. a DAE file). The scene definition file is processed, using a loader of an appropriate 3D type. It does not matter which loader is used (i.e. in which 3D format the models are), the result is very much the same. There are some subtle differences, i.e. some loaders create geometries as instances of `Geometry`, some as instances of `BufferGeometry`, but as we have explained in the previous Section 9.2, apart from an increased memory consumption etc. (and in this case, also increased load time), this does not matter as far as the runtime performance goes if the loader does not optimize the created `BufferGeometry` (and most of the loaders do not optimize at all, if any).

The loader creates a **new instance of `Mesh` class for each collection of faces** which have same material and are connected to each other by edges (more on this later). The process is described through pseudocode on Figure 9.2.

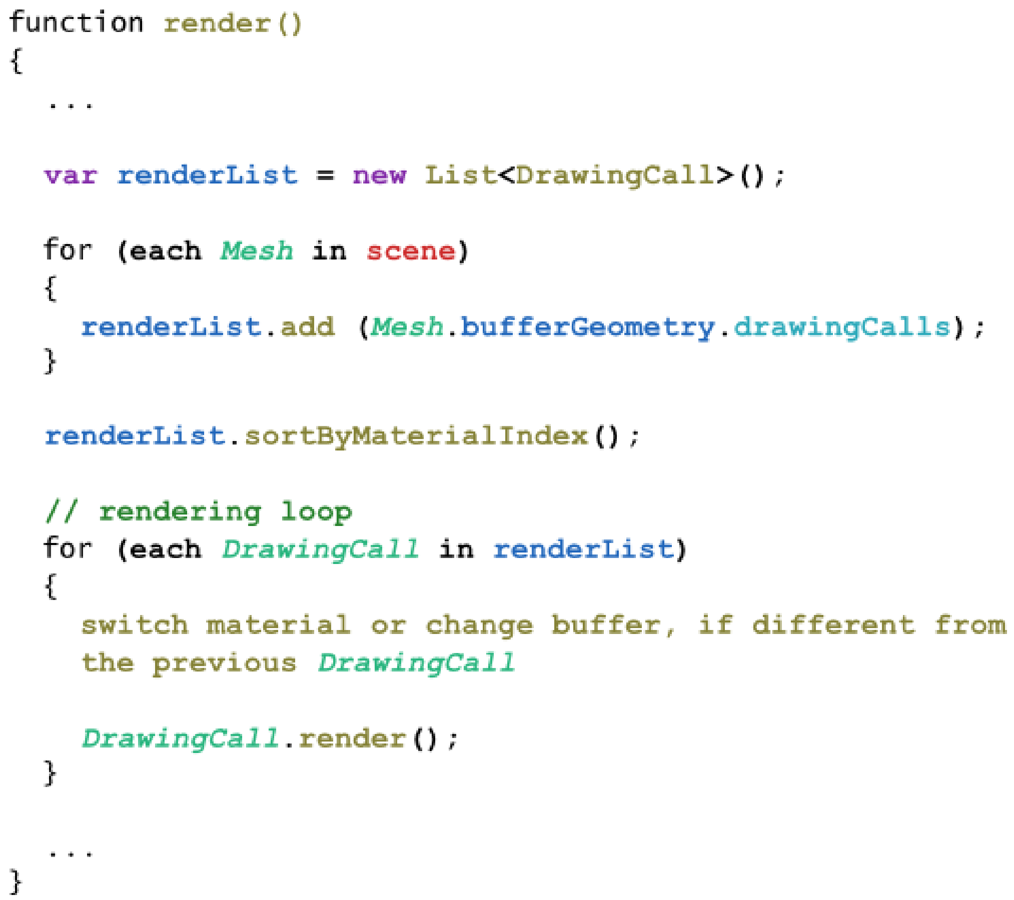


```
function loadSceneNaive (definitionFile)
{
  for (each faceCollection found in definitionFile)
  {
    var objectInformation = parse (faceCollection);
    var mesh = new Mesh (objectInformation);

    scene.add (mesh);
  }
}
```

Figure 9.2. Naive scene loading code

When the loader finishes, the scene contains N meshes. Each such `Mesh` has its own data buffer with the geometry data[1]). How the rendering process looks in Three.js can be seen in the pseudocode on Figure 9.3.

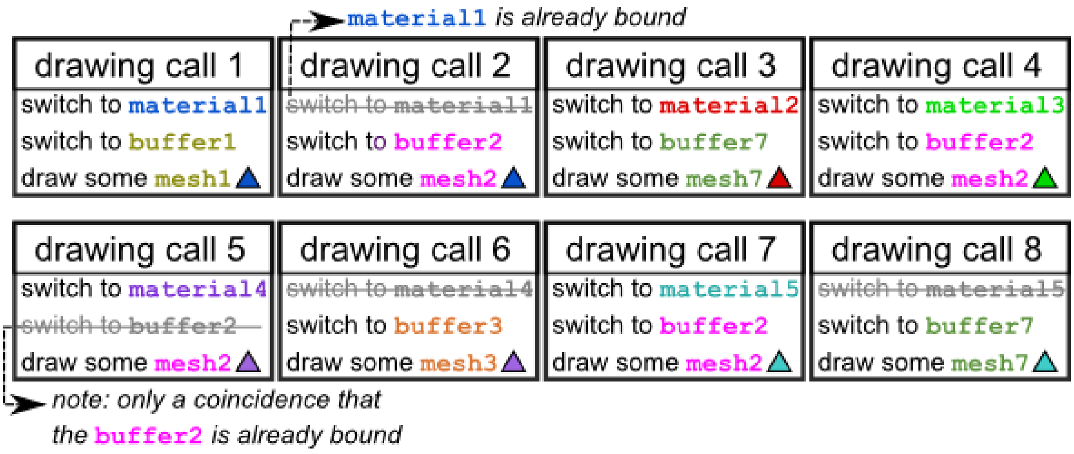
[1]) Actually, it usually has one buffer for each attribute (position, normal, texture coordinates, ...), but we try to keep it simple for our explanation.



```
function render()
{
  ...

  var renderList = new List<DrawingCall>();

  for (each Mesh in scene)
  {
    renderList.add (Mesh.bufferGeometry.drawingCalls);
  }

  renderList.sortByMaterialIndex();

  // rendering loop
  for (each DrawingCall in renderList)
  {
    switch material or change buffer, if different from
    the previous DrawingCall

    DrawingCall.render();
  }

  ...
}
```

Figure 9.3. Three.js render function (part)

Essentially, definitions of drawing calls are collected from each `Mesh`, sorted by materials (to reduce WebGL state changes) and then executed in such order. Example of a sorted render list can be seen on Figure 9.5. This render list is related to the example scene illustrated on Figure 9.4.



material1 *is already bound*

drawing call 1
switch to material1
switch to buffer1
draw some mesh1 ▲

drawing call 2
~~switch to material1~~
switch to buffer2
draw some mesh2 ▲

drawing call 3
switch to material2
switch to buffer7
draw some mesh7 ▲

drawing call 4
switch to material3
switch to buffer2
draw some mesh2 ▲

drawing call 5
switch to material4
~~switch to buffer2~~
draw some mesh2 ▲

drawing call 6
~~switch to material4~~
switch to buffer3
draw some mesh3 ▲

drawing call 7
switch to material5
switch to buffer2
draw some mesh2 ▲

drawing call 8
~~switch to material5~~
switch to buffer7
draw some mesh7 ▲

note: only a coincidence that the buffer2 *is already bound*

Figure 9.5. Render list – naive scene loading

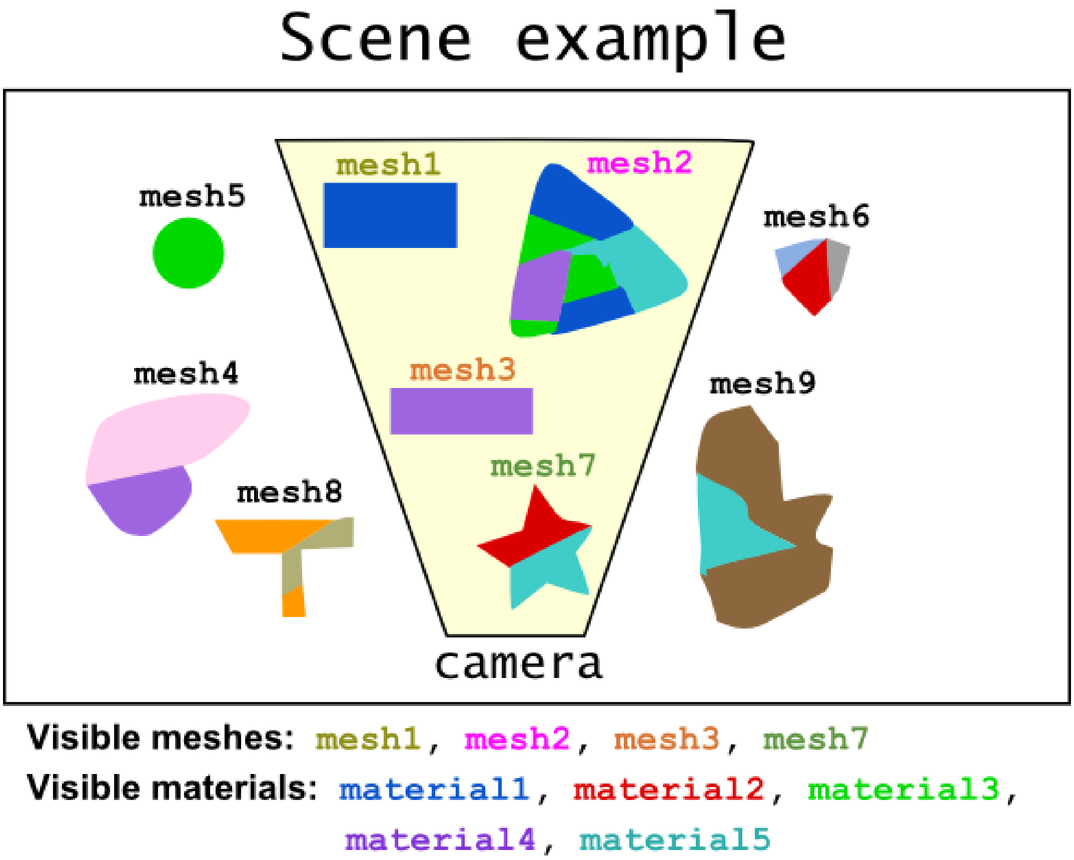

Scene example
mesh1
mesh2
mesh5
mesh6
mesh4
mesh3
mesh9
mesh7
mesh8
camera
Visible meshes: mesh1, mesh2, mesh3, mesh7
Visible materials: material1, material2, material3,
material4, material5

Figure 9.4. Scene example used for demonstration of different approaches to rendering

The breakup of a time complexity for the naive scene loading can be seen on Table 9.1. Buffers have to be switched $O(n)$-times in the rendering loop, because each Mesh has its own buffer[1]), and drawing calls of a Mesh are scattered through the render list. That is a lot of buffer switches, which heavily contributes to the poor performance. Materials are switched just $O(m)$-times, because drawing calls are sorted by materials. There are $O(n)$ drawing calls performed – one draw call for each Mesh. The frustum culling[2]) is performed on each Mesh, but even in the "best" case of camera position and orientation, the application runs at just 25 FPS. Usually, the framerate is around 10 FPS. It is true that performing frustum culling on each individual Mesh without some sort of acceleration data structure is certainly not optimal, but the thing is that even with the culling off (or with the whole scene in the frustum), the performance is about 2 FPS.

From inspecting Table 9.1, we can see that the performance is greatly affected by the number of created Meshes during the import. As for importing, it is really important how a scene is defined and how the loader uses this information. 3D formats usually contain some kind of "group" (alternatively called also "object") definition, which allows us to group faces together to form an object. If the scene definition file contains such group/object definitions in a reasonable amount, then ideally, each found group is imported by the loader into the application as a whole Mesh and all is good, since there

[1]) We once more remind that we made a simplification and assume that each mesh has only one buffer (possibly interleaved), although in practice, it can very well have buffer for each of its attributes; the principles which are explained apply in both cases

[2]) We made a rough assumption that it averagely culls about 50 % of objects

Action	Complexity	Testing scene value
buffer switching	O(n) * f	9 500
material switching	O(m)	140
drawing calls	O(n) * f	9 500
triangles rendered	O(n) * f * trisAvg	152 000
FPS	—	**10 (min 2, max 25)**

Symbol	Meaning	Testing scene value
f	amount of meshes in the camera frustum i.e. visible (normalized to [0,1])	0.5
n	number of all created `Mesh` instances during the scene import	19 000
m	number of materials	140
trisAvg	average number of triangles per `Mesh`	16

Table 9.1. Time complexity – naive rendering approach

are not that lot of groups, so we will not have a lot of `Meshes`, which means no excessive buffer switching and drawing calls.

But, if the definition file does not contain any group definitions or the loader does not support them, we are in for a trouble (which was exactly what happened in the naive scene loading case). The precise result in this unfortunate situation depends on the used 3D file format and the modeling program used for exporting the scene definition file, and in our case with the scene definition file exported by SketchUp, we can generally say that `Meshes` created by the Collada loader are one-material meshes, and we end up with one `Mesh` instance per each *collection of faces* (triangles) which have same material and are connected to each other by edges (i.e. loosely similar to a triangle strip). In the theoretical worst case when no triangles share edges, we can end up with T instances of `Mesh` class, where T = number of triangles in a scene. Figure 9.6 illustrates all these mentioned cases – there are three materials in total, triangles are labeled by a letter and number (e.g. `F1`) and each case have written below itself how many `Meshes` will be created and from which triangles they consist of.

The loader implementation is extremely important here – it must be able to properly read groups from the scene definition file and create corresponding multi-material meshes. Implementation of the Collada loader used for importing our scene does not support groups per se since it does not create multi-material meshes, but at least it respects groups definitions made in SketchUp. All faces which belong to the same group in SketchUp are imported as one-material meshes (in the manner discussed a moment ago) and become children of an `Object3D` instance, which in this case can be thought of as a container holding multiple `Mesh` instances together. The fact that we know which meshes belong together and are forming a group will be useful a while later when we will discuss improvement of the loading process.

9.3.2 Improved rendering approach – Mesh merging

To summarize, the bottlenecks of the naive scene loading process are:

- state changes caused by switching buffers
- number of drawing calls

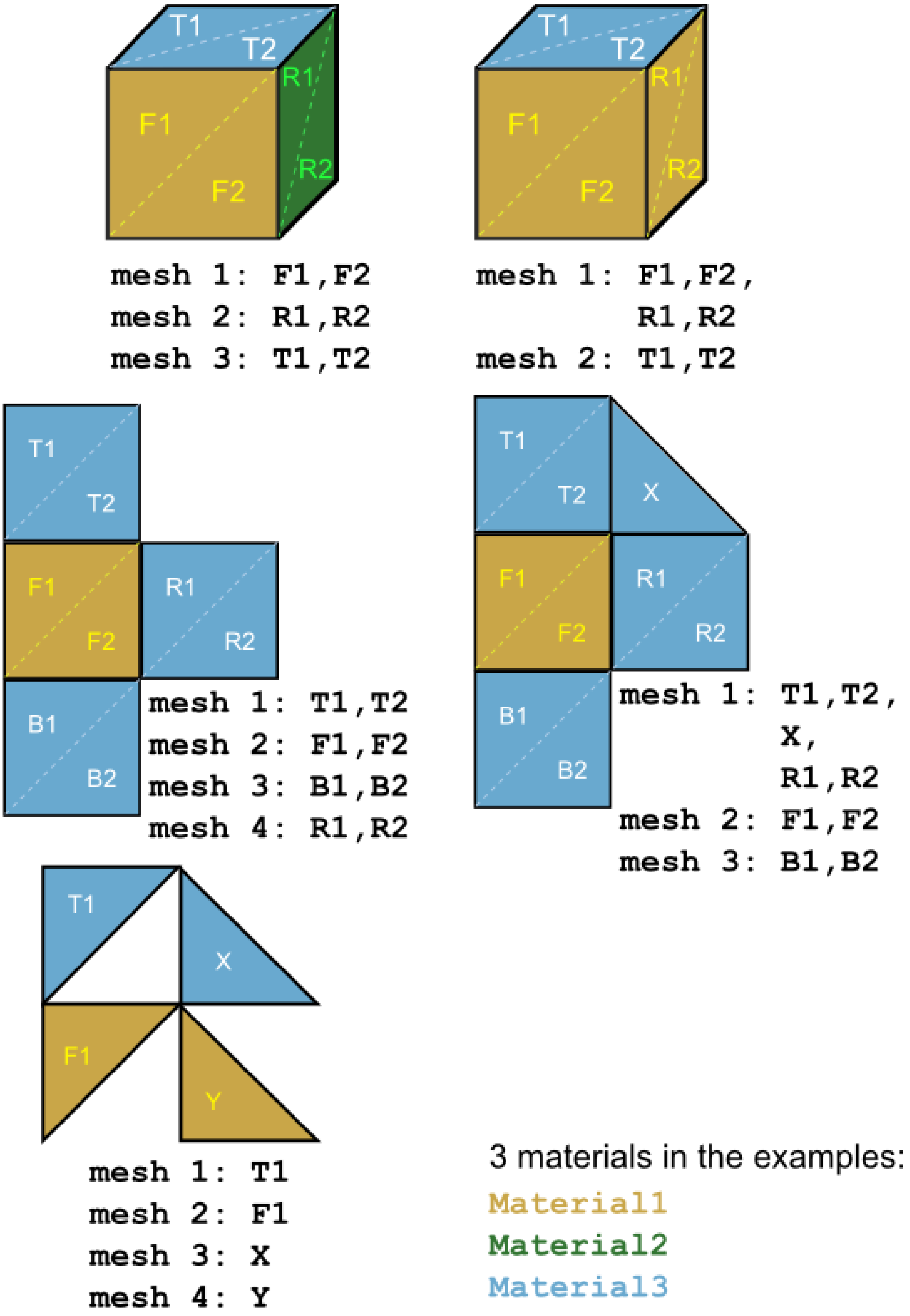


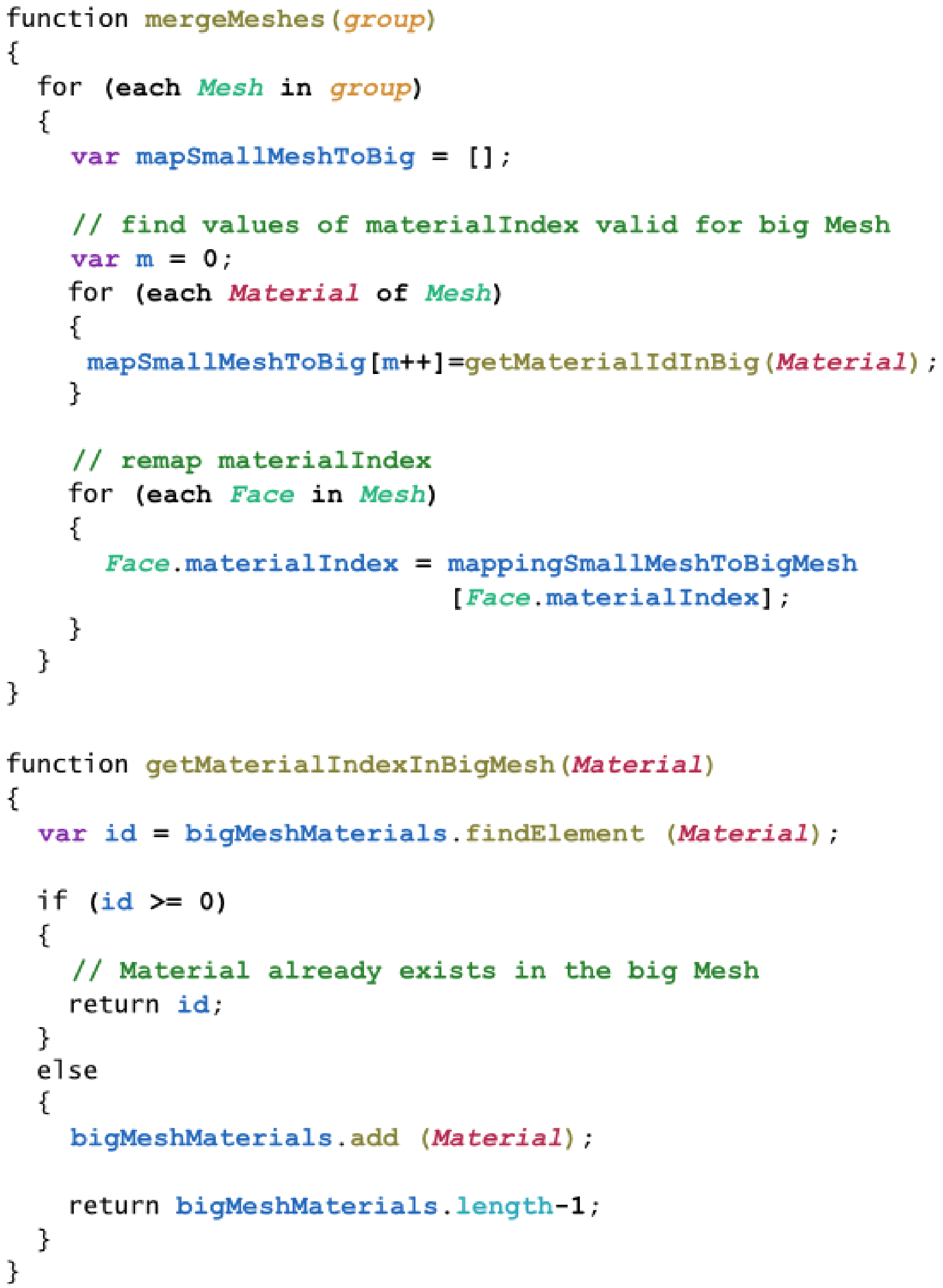
Figure 9.6. Examples of created meshes – naive rendering

Since there were so many meshes, the amount of both shot through the roof, making the performance so bad. Lets look at the way how to fix **the problem of buffer switching** first.

We have talked about the Collada loader and the fact that it does not create a multi-material Mesh representing a group defined in the scene definition file. However, it creates object groups in a form of Object3D container which contains all meshes that belong to the group. The obvious improvement is to create a single big Mesh representing each group (as was always meant to be), which would dramatically reduce number of buffer switches. The big Mesh is multi-material, so it is possible to define different materials for different faces, allowing us to store geometry data of each Mesh from a group inside buffer(s) of the big Mesh.

But for now, in order to better explain the process of creating such single big Mesh representing a group, suppose that we do not have any information about object groups, i.e. the scene definition file contains no object groups at all. That is not an unrealistic scenario at all, considering that working with groups in SketchUp can be quite troublesome. We still want to create a single big Mesh, this time it will represent a whole scene and contain every imported Mesh.

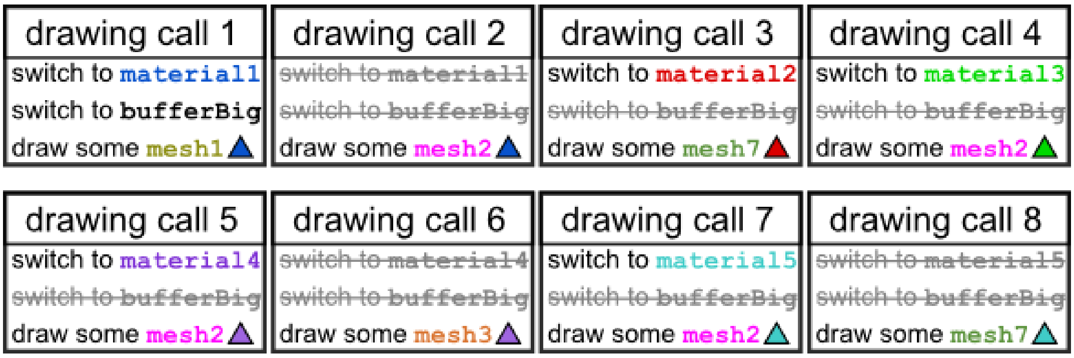
The technique of adding geometries of different meshes together is called the "**Mesh merging**" (also known as the "Mesh combining") and is very useful when dealing with static geometry, which happens to be our case. In the following text, we will refer to the Meshes which are being merged together as "small" Meshes, and the resulting merged Mesh from all these will be referred to as the "big" Mesh. Three.js provides a built-in function for merging Meshes, but we cannot just call it as it is; in brief, the function does not account for the fact that materials occurring in Meshes repeat (e.g. there are 19 000 Meshes, but only 140 materials), making it impossible to correctly remap materialIndex of faces (recall Figure 9.1) of a small Mesh which is currently being merged into the big Mesh. The materialIndex must be remapped, because it is valid only in the context of a small Mesh (and its own list of materials), and we need it to be valid in the context of the big Mesh (and its list of materials). As complicated as it may sound, the solution is actually quite simple. How the whole Mesh merging procedure could be carried out is described in the pseudocode on Figure 9.7.



```
function mergeMeshes(group)
{
  for (each Mesh in group)
  {
    var mapSmallMeshToBig = [];

    // find values of materialIndex valid for big Mesh
    var m = 0;
    for (each Material of Mesh)
    {
     mapSmallMeshToBig[m++]=getMaterialIdInBig(Material);
    }

    // remap materialIndex
    for (each Face in Mesh)
    {
      Face.materialIndex = mappingSmallMeshToBigMesh
                              [Face.materialIndex];
    }
  }
}

function getMaterialIndexInBigMesh(Material)
{
  var id = bigMeshMaterials.findElement (Material);

  if (id >= 0)
  {
    // Material already exists in the big Mesh
    return id;
  }
  else
  {
    bigMeshMaterials.add (Material);

    return bigMeshMaterials.length-1;
  }
}
```

Figure 9.7. Mesh merging – pseudocode

After the remapping of `materialIndex` is finished in all small `Meshes`, we can safely merge them together into the one big `Mesh` using the built-in Three.js function. For example how a sorted render list can look now that we have all meshes' data in one buffer, see Figure 9.8 (same as with the previous example of a render list, this render list is also related to the example scene on Figure 9.4).



drawing call 1
switch to material1
switch to bufferBig
draw some mesh1▲

drawing call 2
~~switch to material1~~
~~switch to bufferBig~~
draw some mesh2▲

drawing call 3
switch to material2
~~switch to bufferBig~~
draw some mesh7▲

drawing call 4
switch to material3
~~switch to bufferBig~~
draw some mesh2▲

drawing call 5
switch to material4
~~switch to bufferBig~~
draw some mesh2▲

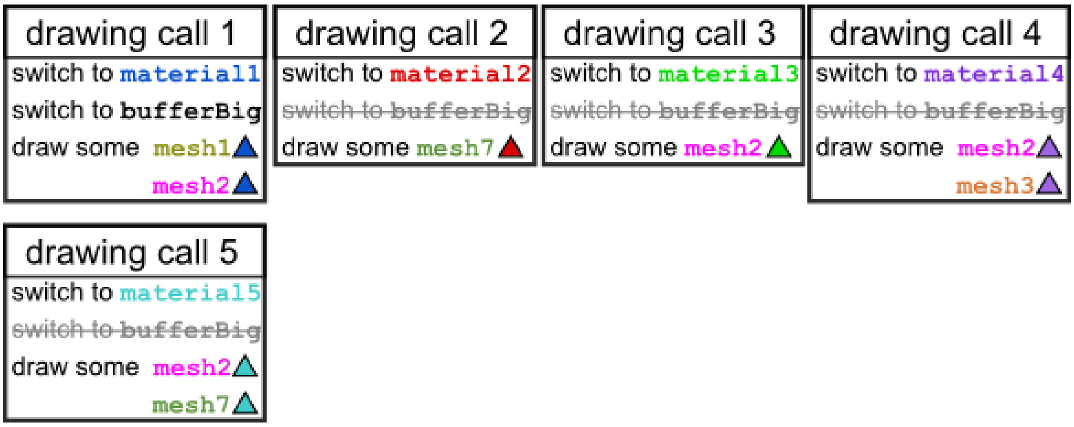
drawing call 6
~~switch to material4~~
~~switch to bufferBig~~
draw some mesh3▲

drawing call 7
switch to material5
~~switch to bufferBig~~
draw some mesh2▲

drawing call 8
~~switch to material5~~
~~switch to bufferBig~~
draw some mesh7▲

Figure 9.8. Render list – Mesh merging

Lets move to the solution of **the second problem with a high number of drawing calls**. During the merging of small `Meshes` into the big `Mesh`, geometries of small `Meshes` were just being added one after another into the buffer of the big `Mesh`. If we take a closer look at the example of a sorted rendering list obtained after the Mesh merging, we see that except for the occasional and necessary material switches, there are no other state changes happening between the drawing calls. It is thus possible to draw all triangles of the current material in a single drawing call. However, it is not possible in our current situation, as geometries of triangles with same materials are scattered in the big `Mesh` buffer – we need triangles to be stored one after another. **After we perform triangle rearrangement and have triangles of same materials in contiguous blocks of memory, we are able to draw all triangles with same material at once in a single drawing call**. How the sorted rendering list from previous examples (see Figure 9.5 and 9.8) looks now can be seen on Figure 9.9.



drawing call 1
switch to material1
switch to bufferBig
draw some mesh1▲
mesh2▲

drawing call 2
switch to material2
~~switch to bufferBig~~
draw some mesh7▲

drawing call 3
switch to material3
~~switch to bufferBig~~
draw some mesh2▲

drawing call 4
switch to material4
~~switch to bufferBig~~
draw some mesh2▲
mesh3▲

drawing call 5
switch to material5
~~switch to bufferBig~~
draw some mesh2▲
mesh7▲

Figure 9.9. Render list – Mesh merging with triangle rearrangement by materials

To further illustrate the situation, we provide another example on Figure 9.10. Note that the example operates on a different scene than previous examples; only objects visible to the camera are depicted (the rest of the scene is omitted), the white numbers express how many triangles of a mesh are drawn in a corresponding material, e.g. `mesh1` consists of 8 blue (`material1`) triangles, 4 red (`material2`) triangles and 66 green (`material3`) triangles. Next, the rendering list along with `DrawingCall` definitions is shown. `DrawingCall` is a class which instances are used for defining a single WebGL drawing call – `count` defines how many triangles will be drawn, `start` tells us from which *item* (“triangle”) in a buffer should drawing start and is used to compute the actual

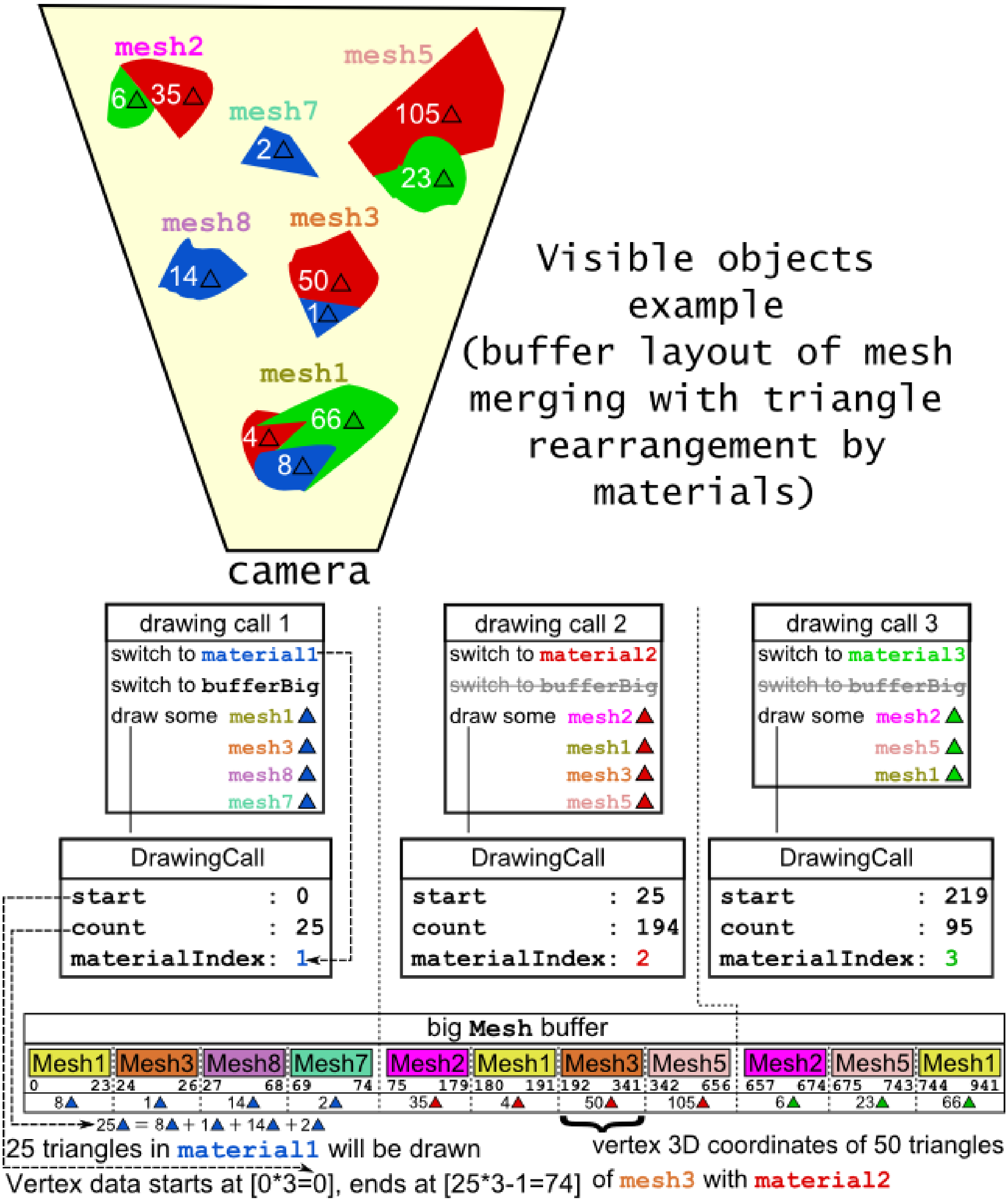


Figure 9.10. Mesh merging – layout of big Mesh buffer

array index from which the data will be fetched (`index = item * itemSize`; `itemSize` being 3 for positions and normals, 2 for texture coordinates etc.), `materialIndex` defines which material will be used for the rendering. At the bottom, layout of the big `Mesh` buffer for vertex coordinates is depicted. If the big `Mesh` contains buffers for another attributes, their layout would be exactly the same e.g. for normals and in the case of texture coordinates, we would multiply by `itemSize` of 2 instead of 3 to compute the array indices.

The explained principle holds even for the case when the big `Mesh` contains one big interleaved buffer with all attributes being together, interleaved e.g. like

`VVVNNNTT|VVVNNNTT|VVVNNNTT|...` (where `V` = vertex position, `N` = normal, `T` = texture coordinate). The only different thing in case of interleaved buffers is a slightly more complex and error-prone implementation, since the data rearrangement during all the Mesh merging becomes trickier. Our application actually uses **one interleaved buffer for all attributes** [1] rather than one buffer for each attribute, but we have chosen not to include interleaved buffer in our explanations for the sake of simplicity.

The technique of Mesh merging, however, has its disadvantages as well, all coming from its very nature. Above all, it is suitable for world containing mainly static geometry which does not change (or changes very rarely); our application fits that perfectly. But what is a problem that we end up (after this second phase – triangle rearrangement) with a single instance of `Mesh` class, which makes some standard features of Three.js (or of any other graphical framework/engine) noticeably harder or impractical to use. One of the most painful case of this is frustum culling, which works with individual `Meshes` (or their groups).

Time complexity is depicted on Figure 9.2. By merging meshes together, we have reduced the time complexity of buffer switching to $O(1)$, effectively eliminating it. Number of material switches is still $O(m)$ and by rearranging the individual triangles by material, we have been able to cut the overall number of draw calls also to $O(m)$. The frustum culling is not performed as we do not treat `Meshes` individually anymore, so all triangles in the scene are rendered. The result is 60 FPS, which is a huge improvement and certainly an acceptable value for us.

Action	Complexity	Testing scene value
buffer switching	O(1)	1
material switching	O(m)	140
drawing calls	O(m)	140
triangles rendered	O(n) * trisAvg	304 000
FPS	—	**60**

Symbol	Meaning	Testing scene value
n	number of all created `Mesh` instances during the scene import	19 000
m	number of materials	140
trisAvg	average number of triangles per `Mesh`	16

Table 9.2. Time complexity – improved rendering (Mesh merging)

[1] Data from our scene loader is not interleaved; and since we always have to move data around during the Mesh merging process anyway, we make use of this opportunity to make the data interleaved

9.3.3 Mesh merging – several groups

Performance of 60 FPS achieved by the Mesh merging is satisfying, but the fact that we cannot use frustum culling may bother us. We still have the geometry data of the individual meshes (and their bounding volumes), so it would be certainly possible to modify Three.js to frustum cull individual small `Meshes` (or even their subset visible in the frustum, obtained by traversing some acceleration data structure) which were merged into the big `Mesh`, but it is not very practical to do so if we want to stick to the triangle rearrangement, because then we could not fully utilize its power (data of all drawn triangles would no longer be in contiguous blocks of the memory).

Now, lets lift our assumption that there are no object groups defined in the scene definition file. More effective and simpler solution to enable frustum culling is to use group definitions from the scene definition file and create a big `Mesh` for each group. That way, we would be able to use frustum culling on each big `Mesh` and still profit from the Mesh merging and triangle rearrangement optimizations, albeit not for free. Figure 9.3 shows time complexity of this approach.

Action	Complexity	Testing scene value
buffer switching	O(1)	1
material switching	O(m)	140
drawing calls	f * O(g) * O(m)	700
triangles rendered	(O(n) * trisAvg) - ((1-f) * O(g) * trisGroupAvg)	152 000
FPS	—	**40**

Symbol	Meaning	Testing scene value
f	amount of meshes in the camera frustum i.e. visible (normalized to [0,1])	0.5
g	number of object groups in the scene	10
n	number of all created `Mesh` instances during the scene import	19 000
m	number of materials	140
trisAvg	average number of triangles per `Mesh`	16
trisGroupAvg	average number of triangles per group	30 400

Table 9.3. Time complexity – improved rendering (Mesh merging, several groups)

Buffer switching can stay at $O(1)$ if we make some modifications to Three.js, store geometries of all big `Meshes` in one huge buffer (which does not belong to any `Mesh`) and then just make sure that each big `Mesh` uses that buffer. The price to pay for the ability to use frustum culling lies in an increased number of drawing calls: each group in the frustum uses $O(m)$ drawing calls. Number of rendered triangles depend on the number of triangles which do not belong to any group (these are rendered every time, since they cannot be culled) and on the sum of triangles of groups in the frustum.

The performance obviously depends most on the value of g, which is responsible for the increased drawing call count. We have performed a benchmark in the testing scene to see how much overhead an increased number of drawing calls pose. Mesh merging with the triangle rearrangement (see Figure 9.2) was used to create a single big `Mesh` representing the whole scene for this benchmark. The results are shown on Figure 9.11 and 9.12. The performance began to significantly drop down after about 400 draw calls on the machine which ran the benchmark (see its HW specification in 11.2 – Desktop 2007). Naturally, this number will be different for each machine, but since the hardware specification of the testing machine qualifies even below the worst configuration which should still be able to run the application, we can consider the number to be quite reliable for the general orientation.

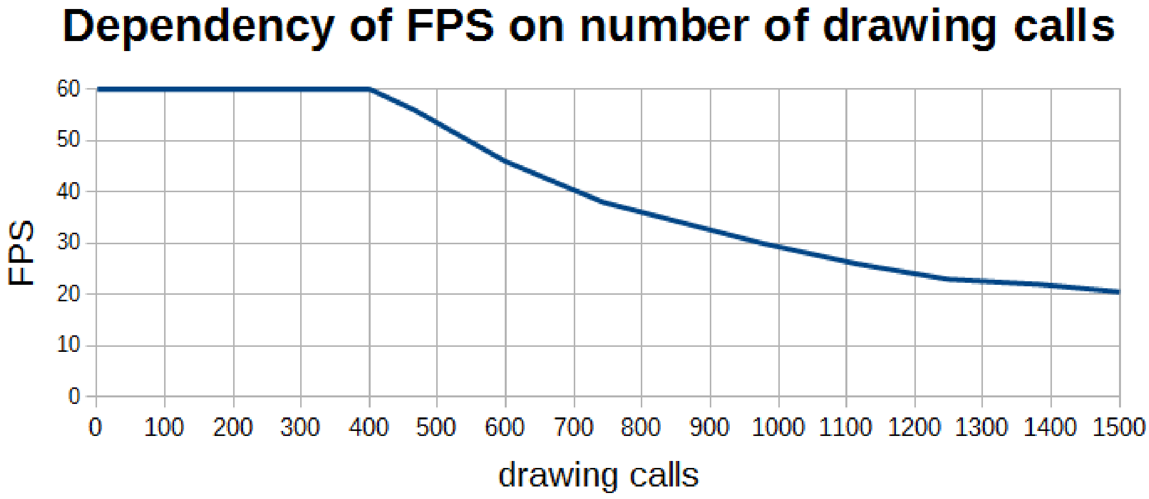


Figure 9.11. Dependency of FPS on drawing calls

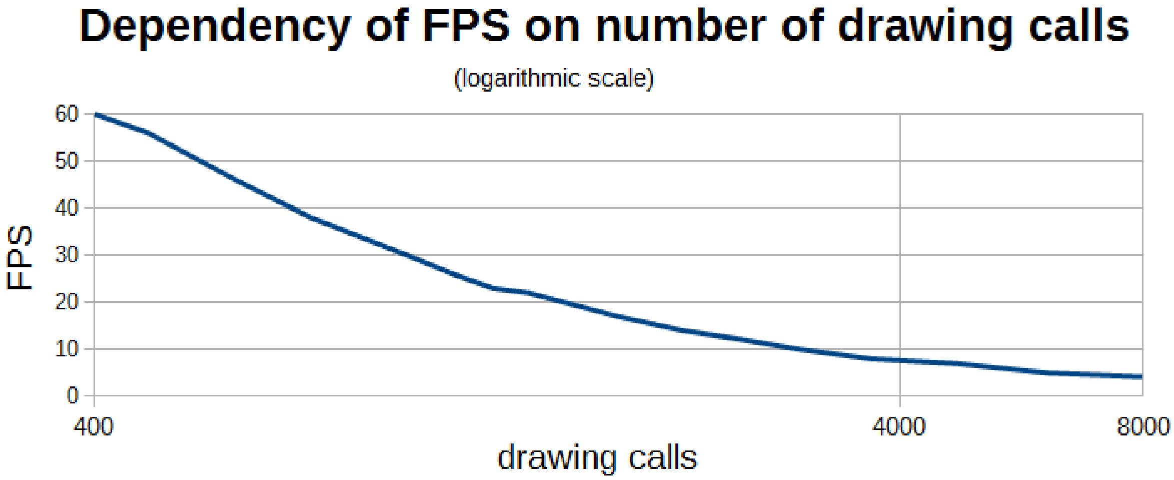


Figure 9.12. Dependency of FPS on drawing calls (logarithmic scale)

We have to keep the rough limit of approx. 400 draw calls in mind if we decide to use the frustum culling. A reasonable solution is to keep the number of groups low (so we do not reach the limit) and create a group only if the resulting big `Mesh` would have “a lot of triangles”. Also, small `Meshes` which form a group (in other words, small `Meshes` which form a big `Mesh`) should be close to each other, so there is a realistic probability that the big Mesh could be culled; if small Meshes are too far apart, the bounding volume for the entire big Mesh (group) is too large, making our efforts vain.

What exactly is "a lot of triangles" is not easy to define, it depends on the concrete application and shall be tested. In any case, it must be profitable to create a group – if we create a group for a fountain which has 1 000 triangles, we may be able to cull it sometimes or even display with a lower LOD, but it would not be worth the added extra drawing calls which fountain mesh will produce. On the other side, if a fountain has over 100 000 triangles and a complicated shader, then it may be worthy of consideration.

As for our case: the basis for creating our testing scene (300 000 triangles, 140 materials) was an unfinished version of the scene appearing in the application. The unfinished version of the scene covers about 25 % of the Prague Castle area and contains around 20 000 triangles and 60 materials. We can estimate that even the whole Prague Castle scene will not exceed 300 000 triangles and about 300 materials. It seems that creating groups (and using the naive Three.js built-in frustum culling + LOD) in our current situation is not worth it and may be even counterproductive – because solely by the estimated number of materials, we are already approaching the draw call limit for sustaining 60 FPS. On the contrary, the estimated number of triangles is similar to our testing scene and thus "safe", so currently there does not seem to bee the need to "save on" triangles. On that account, we stay with a solution of creating a single big Mesh for the whole scene while employing triangle data rearrangement, i.e. the approach that was exactly explained in 9.3.2).

Another alternative approach which we want to mention is giving up the whole triangle rearrangement optimization which forces us to have a single big Mesh and instead implement a more sophisticated culling process using an acceleration data structures like kD-tree or Bounding Volume Hierarchy. Inseparable part in this approach would be incorporating LOD as well, because even the best culling techniques are not of much use if the avatar is in the FLY mode and the whole castle area is visible. However, keeping the conclusion from the previous paragraph about the whole castle scene in mind, we think that **our application with its intended scale and graphics features performs good enough with the triangle rearrangement**.

9.4 Possible performance improvements in the future

The larger value of the estimated number of materials in the whole scene (see the end of the previous section 9.3.3) may concern us – after all, we can say that the performance of our application depends mostly on the number of drawing calls, which in our case equals number of materials. Besides, the limit of 400 drawing calls is just an approximation and apart from the sun and real-time shadows casted by it, the testing scene which was used to derive this limit did not contain any effects or light sources, so the real limit is certainly smaller. Moreover, if we want to, then exceeding "safe" parameters of the scene (300 000 triangles, 300 materials) is a piece of cake. We will now quickly discuss how to deal with this issue to some degree.

Concerning the material limit, it is important to note that almost each texture in our application corresponds to a single image, no matter how small/big it is. This is a very convenient during modeling (i.e. no pre-processing and easier manipulation), but of course not the most effective way of handling textures, since when we use a whole 4096×4096 texture (i.e. one whole material) on a 256×256 image, then 99,61 % of texture's area is left unused. Enter the technique of **"Texture atlasing"**, which can help us to pack several images into one texture. Although there would still be some unused area in textures, their overall number would be hugely decreased. The sheer number of materials (textures) would no longer be a significant problem.

While the texture atlasing is an effective optimization, we still have to keep in mind available memory on graphics card. Even if the number of materials (textures) may not be a problem, the total image area covered by them still concerns us. Luckily, we can use a **texture compression algorithm** like S3 (probably more known by the name of DXT) [53], which is able reduce size of a texture on GPU about 4×.

Chapter 10
Lighting and visual improvement techniques

Having nice 3D models and textures in virtual worlds is crucial for a good impression, but not enough on its own. Simulation of lighting has been always known as probably the number one thing which brings a virtual scene closer to the reality and improves its visual quality. We will briefly talk about illumination models and how they simulate lighting and shadows, pinpoint their pros and cons and describe how the whole simulation is done in our application which contains a day/night cycle. However, there are also other ways how to improve the visual perception of a scene; we will particularly focus on post-processing effects.

10.1 Goals

Our aim is to improve the visual quality of our application and making it appear more realistic, yet still running at smooth speed. We shall:

- decide on a suitable illumination model and algorithms
    - shall be applicable for a real-time rendering
    - shall involve little to none pre-processing work
    - keep in mind integration with the rendering engine (Three.js in our case) and its limitations
- choose a technique for including shadows
    - it should adapt to a real-time change of lighting conditions
- employ methods for reducing rendering artefacts (aliasing etc.)
- incorporate graphics techniques that can further improve the visual experience

10.2 Illumination models

Illumination models are used to generate the color of an object's surface at a given point on that surface – it specifies how to compute intensity of light with respect to the observer. The factors that govern the illumination model determine the visual representation of that surface. It usually considers light attributes (position, color, intensity, ...), surface properties (color, reflectivity, ...) and interaction among lights and objects [54–55]. We can divide illumination models (techniques) into the two major categories:

- Local
    - only takes into the account the light which comes directly from a light source (direct illumination)
        - considers just light sources and surfaces properties

  - modeled by using a Bidirectional Reflectance Distribution Function (BRDF)
    - tells us the ratio of light coming from one direction that gets reflected in another direction
  - no shadows
  - fast to compute
  - e.g. Phong, Cook-Torrance
- Global
  - apart from direct illumination, it incorporates all light vs surface interactions in a scene (indirect illumination)
    - when a ray of light hits a surface, it affects the surface and is further reflected, continuing its travel into another direction until it hits some surface again and the cycle repeats
  - involves tracing of light paths
    - possible to simulate reflections, refractions, caustics, ...
  - hard and soft shadows
  - slow to compute
  - e.g. Radiosity, Ray-tracing, Photon Mapping, Ambient Occlusion

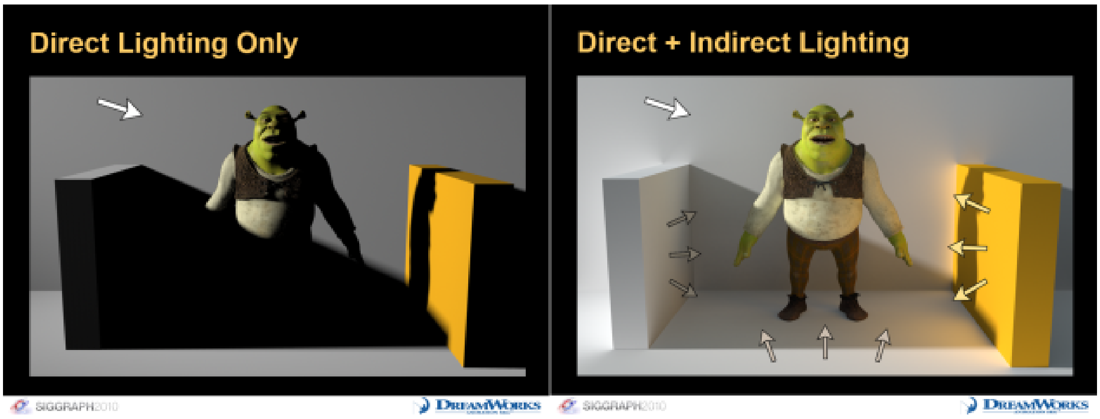


Figure 10.1. Direct vs indirect illumination [56–57]

The difference between a direct and indirect illumination is shown on Figure 10.1. Obviously, a global illumination model is superior as far as the visual quality goes – and unfortunately a lot slower to compute than a local illumination model. The situation is even worse due to the fact that WebGL cannot utilize full machine's hardware potential in contrast to, for example, its OpenGL counterpart. There have been a few experiments involving global illumination and WebGL that are running real-time, e.g. "WebGL Deferred Irradiance Volumes" [58] or "WebGL Path Tracing" [59], but it does not seem that their results would be applicable to a more complex application. "WebGL Deferred Irradiance Volumes" has actually trouble running real-time when rendered in the full resolution on our testing machine (see Desktop 2007 in Table 11.2; running in half of the resolution was smooth, but the visual quality took a hit) and its development involved a lot of pre-processing. "WebGL Path Tracing", although running smoothly, has unacceptable render quality – the image starts off as very grainy (Figure 10.2) and

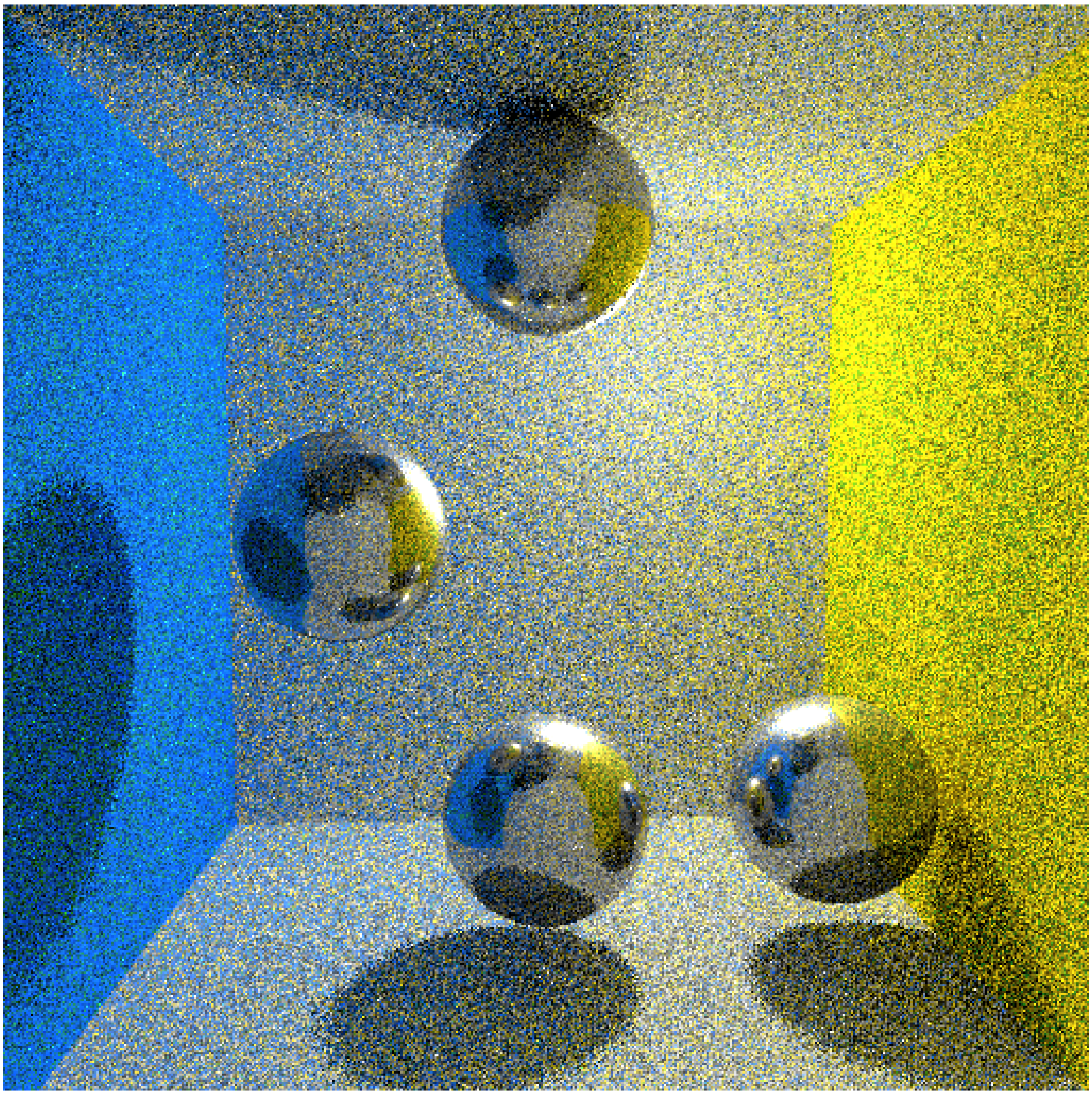


Figure 10.2. WebGL Path Tracing – initial render quality [59]

it takes too long until the noise is cleared and the image starts to looks nice; such approach would not be acceptable in a virtual walk application. And if we consider hardware of machines targeted by our application, we are left with no other choice than to forget global illumination models and stick with local ones, which are fast enough.

In the context of our application and its rendering engine Three.js, we can work with Phong illumination model, which is an empirical model based on observations of its author. This model decomposes BRDF into the following parts [60]:

- Ambient
  - avoids the complexity of global illumination
  - approximates the reflection of all indirect illumination with a constant
- Diffuse
  - assumes ideal diffuse reflectance (i.e. surface reflects light equally in all directions)
  - view independent

- Specular
  - assumes ideal specular reflectance (i.e. surface reflects only in mirror direction)
  - view dependent
  - creates “highlights” on a surface

The final result is sum of these parts and looks acceptable, but the crude approximation of an indirect illumination by the ambient term is not really convincing. Despite the fact that we have declared global illumination models as not being fit for our application, there is actually one among them (with somewhat less performance requirements) which can help us to salvage the situation: the **Ambient Occlusion**. It is a shading and rendering technique used to calculate how exposed each point in a scene is to the ambient light. The ambient occlusion is usually calculated by casting rays in every direction from the surface.

Rays which reach the background increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry are rendered dark, whereas points with little geometry on the visible hemisphere appear light. The main reason for using ambient occlusion is to achieve nice-looking soft shadows, which make objects look real, without the effort of a more complex global illumination model. It is typically stored in a texture map or as vertex attributes [61]. Since we try to steer clear off pre-processing and strive for real-time solution, we use variant of the technique called the **Screen Space Ambient Occlusion** (SSAO). SSAO is performed as a post-processing effect and uses depth buffer values to approximate the scene geometry. It has several advantages over its object-space version [62]:

- no pre-processing
- not affected by the scene complexity
- simpler to implement

The disadvantage of SSAO is that it is not physically correct and sometimes contains artefacts, and the results can be view-dependent. Despite that, it is an improvement for the visual quality of our application. While it is true that SSAO produces some shadows, their impression is limited and not really enough for the impression we want to make. And we want to have nice shadows, for they are an important visual and depth cue and help with understanding of spatial relationships between models (relative positions of objects, light positions, ...) and greatly improve overall feeling from a scene. Unfortunately, they also belong to the family of Global illumination models (techniques).

10.3 Shadows

We have considered two probably the most well-known algorithms for computing shadows, as described in [63–64]: Shadow Volumes and Shadow Mapping.

Shadow Volumes is an object-space algorithm which produces pixel-perfect, hard shadows. First, for each scene object, a Shadow Volume (see Figure 10.3) is constructed such that object’s polygons are extruded to semi-infinite volumes by casting a ray from a light’s point of view. In the rendering phase, after the scene is rendered normally once and the depth buffer is filled, Shadow Volume is rendered twice (once with front-faces culled and once with back-faces culled) and the stencil buffer is modified. Finally, by testing values of the stencil buffer, it is possible to find out which fragments are not in the shadow and render them. The good thing about Shadow Volumes is that unlike with Shadow Mapping, it can handle point lights very well. The disadvantage of Shadow Volumes which is quite relevant for our case is that a shadow have to match geometry of a mesh that casts it. This would be a problem for sprites that approximate complex objects (e.g. tree, statues, ...), since their geometry consists of just one quad, which means the shadows would be incorrect for those. They are also intensive on the fill-rate of the GPU.

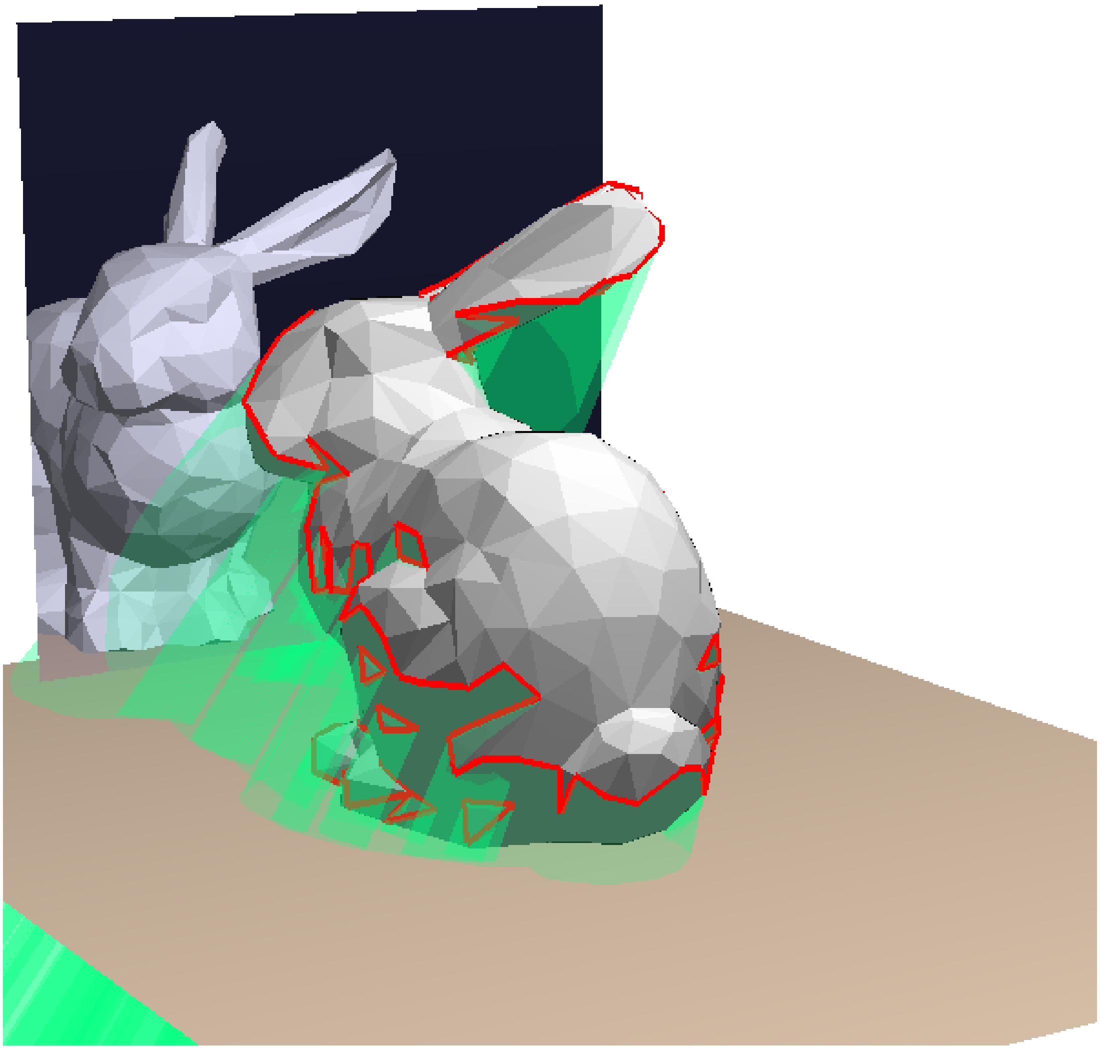
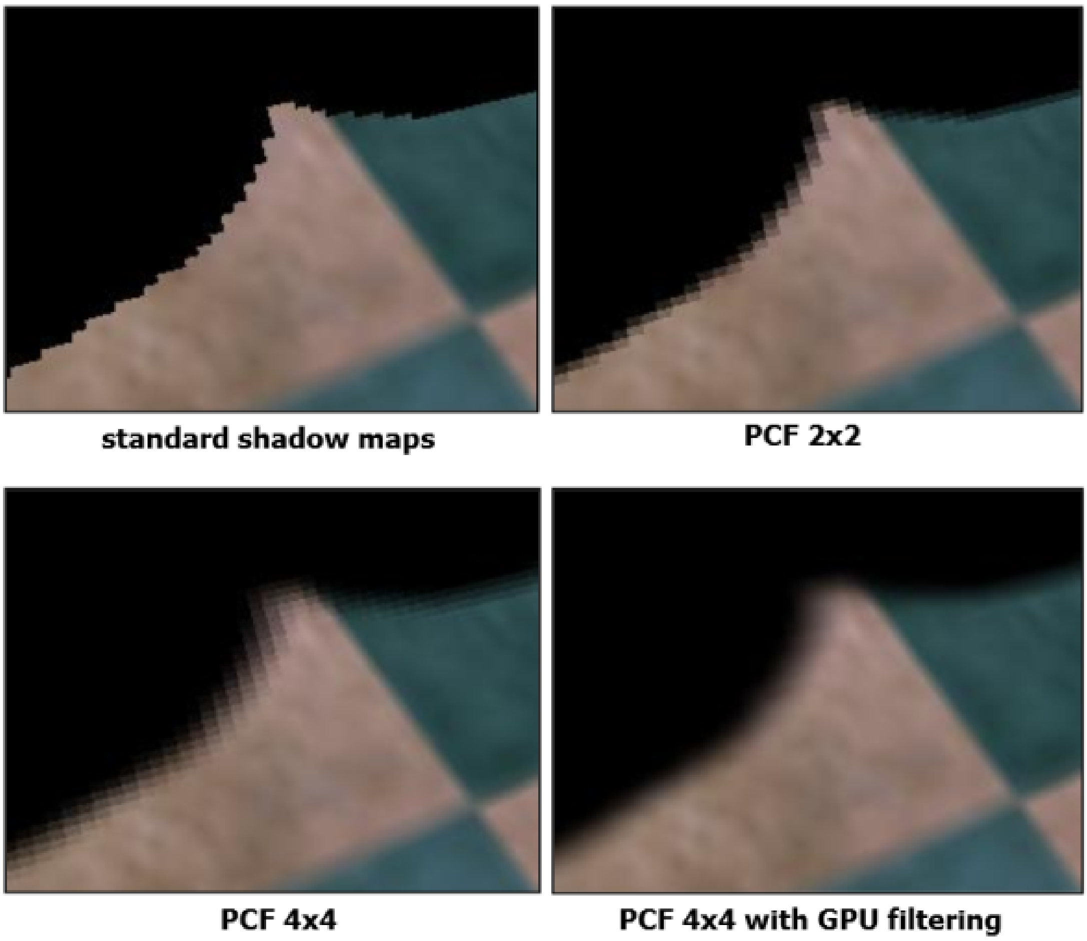


Figure 10.3. Shadow Volumes – visualization [65]

Shadow Mapping is a screen-space algorithm with a simple premise: first, the scene depth map – a.k.a *shadowmap* – is rendered from light's point of view, then the scene is rendered from the current camera's point of view and fragments are transformed into the light space, where we can simply compare depths of the camera and light fragments to see which camera fragments are in shadow and which are not. As usual with screen-space techniques, also Shadow Mapping suffers from the anti-aliasing issues and rendering artifacts. Producing shadows for point lights is also problematic in terms of performance, since 6 shadowmaps have to be created; alternatively some more complicated and faster techniques could be used (e.g. Dual-Paraboloid Shadow Mapping), but they are prone to another issues (quality degradation, distortions, ...). On the bright side of things, Shadow Mapping is fast, no information about the scene geometry is necessary and even soft shadows (though little unrealistic) can be produced by them.

While Shadow Volumes may be demanding to compute (depending on the scene complexity), Shadow Mapping is not affected by a complexity of the scene at all, so with Shadow Mapping, we would not need to worry about the scene becoming bigger and the application slower because of that. An important thing to consider is also the goal to offer a day-night cycle simulation. With the sun constantly moving around the sky, the performance cost for repeated computation of the shadow volumes could be significant. Another considerable advantage of Shadow Mapping lies in the support of correct shadows for sprites. The Shadow Mapping is very GPU-friendly, we really do not care about pixel-precise shadows and on top of that, Three.js provides internal support for Shadow Mapping, so we have decided to incorporate Shadow Mapping over Shadow Volumes as the choice for our shadow computing algorithm. Addressing the disadvantages of the Shadow Mapping: aliasing issues can be remedied a lot by using methods that can filter results of the depth comparisons; the example can be seen on Figure 10.4, where a Percentage Close Filtering (PCF) is depicted. Three.js solution that we use for shadows supports PCF, so all shadows in our application are rendered using this method. We consider higher performance requirements in the case of computing shadow for point lights as an acceptable downside.



standard shadow maps

PCF 2x2

PCF 4x4

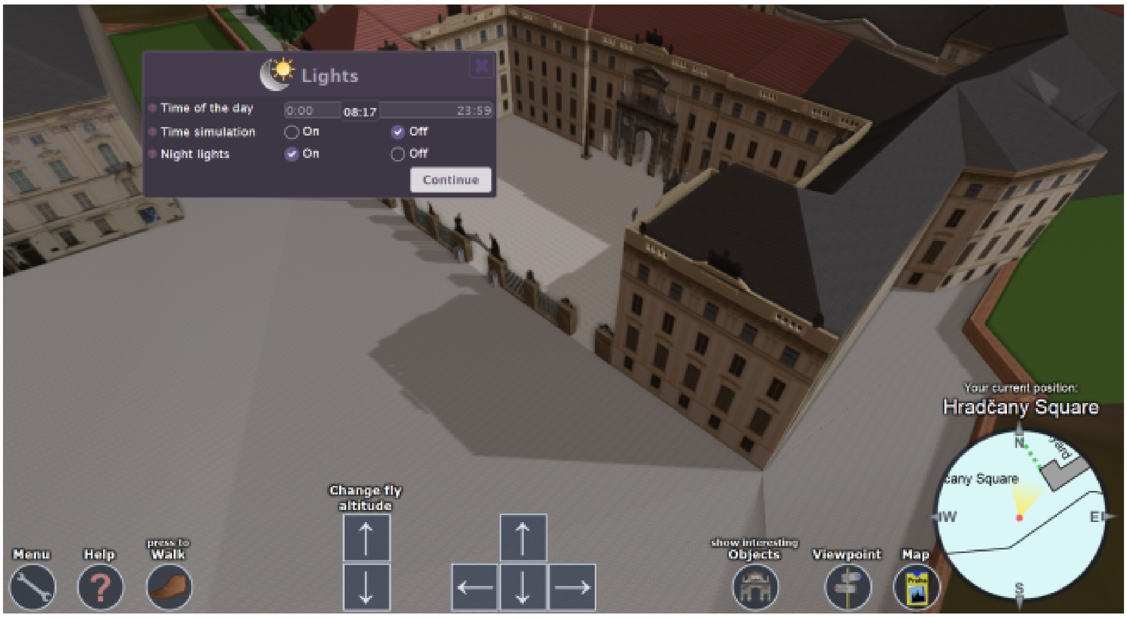
PCF 4x4 with GPU filtering

Figure 10.4. Shadow Mapping – advanced method of Percentage Closer Filtering [64]

10.4 Lighting and other techniques

One of our main goals related to visual effects was to incorporate a day/night cycle into the application. Rendering the **visual effect of sun glare and a corresponding sky appearance** for different positions of the sun is achieved using a shader from [66]. The sun itself is simulated by a **directional light** which has been synchronized with the position of the sun at the mentioned shader. By computing the angle between the sun and the standard up vector $(0, 1, 0)$, we can distinguish between different sun phases – we use this to identify when it is dusk/dawn, day or night. We have enriched the simulation of day/night so that each sun phase has its own **ambient light**; the ambient light of the scene is computed as a linear interpolation between the ambient lights of relevant sun phases for the current sun position, so the transition is smooth.

During the day and dusk/dawn, the only active light is the sun, no other lights are on. When the night comes, the situation is reversed and the lights in the castle area are turned on. We have used **point lights** to simulate the outdoor lighting. Currently, we do not perform shadow computations for these lights, because the impact would not be significant in our scene, so we have decided to save up some performance. To make the whole day/night simulation more believable, we have also incorporated a **lens flare** effect – a light scattering phenomenon in lens systems which produces starburst/ring-like elements on the image. The lens flare is especially prominent when the view is directed at the sun (see Figure 10.6). It is implemented as a post-processing effect using the implementation from [67].



Lights

Time of the day 0:00 08:17 23:59

Time simulation ○ On ● Off

Night lights ● On ○ Off

Continue

Figure 10.5. Shadows via shadow mapping with PCF filter in our application

Another techniques used for improving visual appearance include **bump mapping** and **normal mapping**. Both of these techniques modify the normal vector of a fragment and are trying to create an illusion of a more detailed surface. Bump maps are usually stored as grayscale images since they only represent scalar displacement along the (face) normal, whereas normal maps are stored as RGB images since they store directly values of normal vectors which are used as a displacement. These techniques do not modify geometry at all, only the shading is affected. We use them to improve look of few materials in the application.

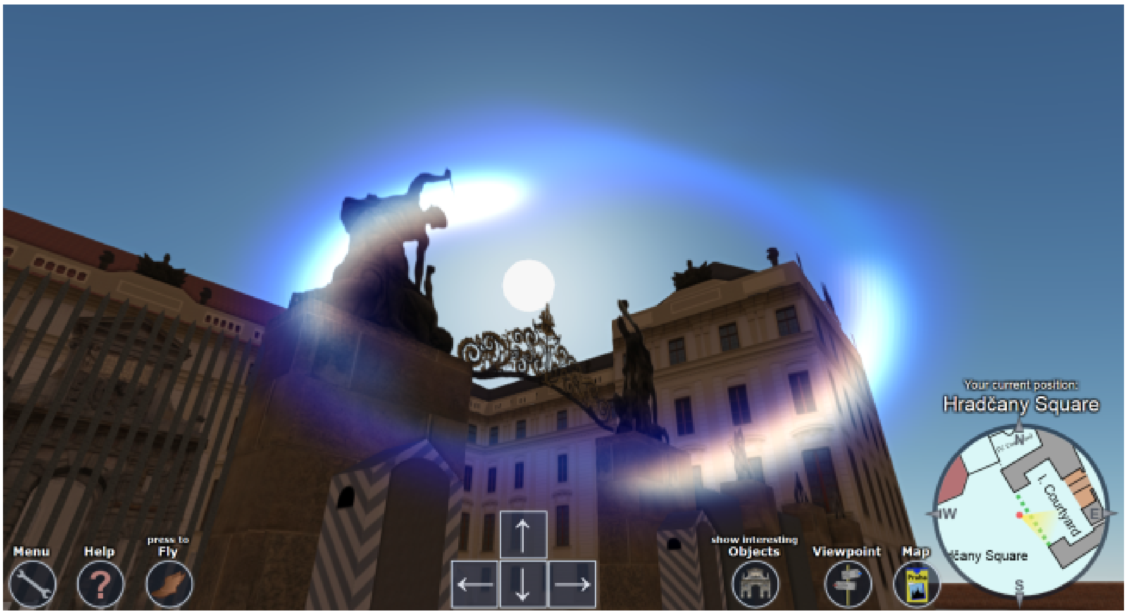


Figure 10.6. Lens Flare effect in our application

10.5 Reducing rendering artefacts

There are situations during the rendering where it is more accurate to talk about *fixing* things rather than improving them. The prime example of such is an everlasting "enemy" appearing in various engineering-related fields – the **aliasing**. Aliasing is an artefact caused by representing a high resolution signal at a lower resolution and happens when the sampling frequency used for sampling the original (continuous) signal is too low. In computer graphics, the most prevalent manifestations of aliasing are jagged lines of rendered models and texture aliasing.

As for texture aliasing, we will have to resample the signal to a different resolution. During texture lookup when it is computed where pixels fall on the texture, it is often the case that one pixel does not correspond to one texel (= no aliasing), but instead:

- one texel corresponds to many pixels
  - when we are close to the texture
  - solved by magnification of texels (enlarging the texture) by "filtering", i.e. computing an average color for a texel using one or more of its neighbors (e.g. nearest-neighbor or bilinear filtering)
- one pixel corresponds to many texels
  - when we are far away from the texture
  - solved by minification of texels (making the texture smaller), again by filtering
  - can introduce aliasing

The problem with filtering is that it can be expensive in terms of performance to filter all texels. An effective solution addressing aliasing and keeping performance in mind is a concept of **Mipmaps**. A mipmap is a collection of one texture in multiple versions, each version is filtered in a different resolution and has half width and height of the preceding version in the collection until it converges to a single pixel, e.g. for a 8×8 texture, the collection will contain 4 versions of the texture: the original 8×8, filtered 4×4, filtered 2×2 and finally filtered 1×1. The further the texture is from us, the smaller version from its mipmap collection is used for filtering. This reduces aliasing and is performance-friendly. The concept of mipmaps is directly supported by WebGL, which is also capable of their automatic generation for a texture upon our request; the quality of automatic generated mipmaps are acceptable, so we use this feature for all textures in the application. It is worth to mention that textures with mipmaps require 33 % more memory, but the results are completely worth it.

Fixing the problem of jagged lines is usually what is meant behind the term anti-aliasing in context of computer graphics. We have mostly two (three) options there [68]:

- Increase the sample rate
    - Super Sample Anti-Aliasing (SSAA)
        - the image is rendered to a much bigger resolution (2×SSAA = 2× bigger, 4×SSAA = 4× bigger, ...) and then downsampled to the displayed size, using the extra pixels for computing the average value for final displayed pixels
        - superior image quality, but terrible performance
    - Multi Sample Anti-Aliasing (MSAA)
        - more performant variant of SSAA – some components of the final image are not fully supersampled
        - smooths object outlines, but not internal areas of polygons
        - we can even request this type of AA to be performed by WebGL, but:
            - no guarantee that the AA will be actually turned on – there is a good chance that browser will not allow AA for older graphics cards to prevent crash of machine; a problem considering hardware which our application targets
            - even if the AA is allowed, it is available only on the main buffer – AA of offscreen buffers or textures is not possible; that is a problem as well, since if we are doing any post-processing (which we do – e.g. SSAO), we will be left with no AA
    - ...
- Use a post-processing effects to blur jagged edges
    - Morphological Anti-Aliasing (MLAA)
        - detects edges (either using color or depth information) and then finds specific patterns in these
        - blends pixels in the edges intelligently, according to the type of pattern they belong to and their position within the pattern
    - Fast Approximation Anti-Aliasing (FXAA)
        - similar to MLAA
    - ...

The post-processing anti-aliasing approaches are comparable to the quality of 4×MSAA while running considerably faster, which makes them ideal candidates for our application. Based on the conclusion from series of benchmarks [69] comparing MSAA with MLAA and FXAA – where MLAA and FXAA produce images of comparable quality, but MLAA is on average 37,1 % slower than FXAA – we have decided that FXAA would be the best solution and **incorporated FXAA implementation** available for Three.js from [70] into our rendering process.

Chapter 11
Testing

This chapter describes testing of the final application and is divided into the two parts. The first part of the chapter focus on the user testing, which involved observation of the users trying to complete a given scenario and a discussion of their observations and opinions. The second part is devoted to the performance testing, where the application was tested on various machines and web browsers.

11.1 Goals

Regarding the user testing, our goal is to find out any flaws in the application's user interface and the application in general, gather opinions of the users and suggest possible improvements. The performance testing shall verify that we have accomplished the goal of keeping smooth framerate on the specified target hardware.

11.2 User testing

The moderator presented participants with the testing scenario and instructed them to read the description and follow specified tasks. User testing was performed using thinking-aloud protocol, which is a direct observation method of user testing that involves asking users to think out loud as they are performing a task. Users are asked to say whatever they are looking at, thinking, doing, and feeling at each moment. This method is especially helpful for determining users' expectations and identifying what aspects of a system are confusing [71].

11.2.1 Testing scenario

We have managed to kept the scenario for testing simple, with just a few tasks, however they cover the whole functionality. Each task can be completed using more than just one way in order to let the users put their inventiveness into the effect – we hoped to see if different users will choose different approaches or if some approach will be more prevalent, where users will have trouble etc. The description of the testing scenario follows, with our commentary below each of the task:

- You will control an application developed as diploma thesis project which visualizes the Prague Castle and allows you to go for a virtual walk in the area. Open the application by clicking on the following link: `http://leyfi.felk.cvut.cz/prague-castle-3d`
- **Task 1 (Explore): Explore the area freely.** Do you recognize some of the buildings? If yes, which ones?
  - (our aim is to let the user freely explore the world and control the application without any pressure; also, we would like to know if the created 3D models resemble their real counterparts in users' opinions)

- **Task 2 (Fountain): There is a fountain somewhere in the area. Find out its name and how old it is.**
  - (we intend to evaluate how the user is able to navigate in the virtual world when searching for something specific; the task is also focused on interaction between the user and an object of interest)
- **Task 3 (Supervisor): Find name of the thesis supervisor.**
  - (we want to know if the user interface is well-arranged enough for the users to find such information)
- **Task 4 (Rooftops): Manage to get on the rooftops.**
  - (we hope to see if the users are able to perform more difficult task and how they will go about it)

11.2.2 Results

A total number of 3 users participated in the testing. One average user, one advanced user and one user who does not use computer very much. It goes without saying that participants do not work in IT related fields, and only one participant had a serious experience with playing 3D games. The general information about the participants can be seen on Table 11.1. A description of the course of the testing in each of the 3 testing cases follows:

Name	Gender	Age	IT/3D skills
Participant 1	Female	24	office work
Participant 2	Male	33	3D games
Participant 3	Male	70	email, news

Table 11.1. User testing – information about participants

Participant 1 became quickly accustomed to the controls; she was using only buttons in the HUD and never used the mouse look. In the Task 1 (Explore), she immediately identified the First Courtyard based on the statues of the battling giants. During the Task 2 (Fountain), she used the Big map option to find out where the fountain is and teleported her avatar to the fountain's location, then she clicked on the fountain and completed the task. The Task 3 (Supervisor) was successfully completed as well – she opened the Help dialog using the HUD button and although she only skimmed over the text, she found the passage mentioning the author and the supervisor and used the provided link in this part to open the Credits section which contains the desired information. At the beginning of the Task 4 (Rooftops), she was wondering how can she do such thing, but quickly spotted the HUD button for turning on the flying mode; however, since she has not been using the mouse look or keyboard (she later said she did not know such thing exists), she was unable to take off the ground and thus did not complete the task. She then continued to explore the area for a few minutes, which was enough time for the day/night cycle to get into the night time; she especially liked the appearance and atmosphere during the sunset. In the discussion, she voiced her negative opinion about the flying controls and suggested that it shall be possible to control the flying in the same way as walking – using the buttons in the HUD. She praised the arrangement of the HUD, simplicity of the controls and clearly visible Help button. She pointed out that when the avatar is looking into the sun, the captions above the buttons[1]) are not visible because of the bright background.

Participant 2 was the only participant who had any serious experience with 3D and games. This background immediately shown itself in the way how the participant controlled the application – he was exclusively using the keyboard and mouse for the navigation of the avatar. He did not recognized any buildings because he does not remember how the castle looks. In the Task 1 (Explore), after a few steps and looking with the mouse around, he quickly clicked on each HUD button and explored the content behind it (Main Menu, Help section, Viewpoint Dialog, Big map). He praised the image of the keyboard in the Help section where controls are written above the keys. In the General settings of the application, he turned on the strafing with the keyboard arrows and increased the mouse sensitivity. It is no surprise that he easily found the fountain by navigating in the virtual world and clicked on it to complete the Task 2 (Fountain). He also pointed out that he thinks that minimap should be interactive in a sense that if he clicks on it, the big map appears. In the Task 3 (Supervisor), he opened the Main Menu and the section with Credits in just a few seconds, completing the task right away. The Task 4 (Rooftops) was completed without any problems as well; the participant also found out that he can walk on the rooftops and spent a little while entertaining himself jumping from one roof to another, and said that the jump feature is the most exciting from the entire application. In the discussion, he said that he liked the controls and especially the ability to jump and fly. He also positively commented on the graphics and said it is a shame that the entire castle area is not modeled and it is evident that some models were modeled better than others.

Participant 3 was the oldest participant in our testing and at the same time had the least experience with computers in general – he generally uses computer just for email communication and reading news. He was able to control the avatar using the arrows on the keyboard, but when he discovered that he can click on the arrows in the HUD, he abandoned the keyboard for the rest of the session, commenting that it is more comfortable for him to handle everything with just mouse. As for the Task 1 (Explore), the participant started with exploring the area of the First Courtyard and then walked into The Second Courtyard. He identified all buildings and objects along the way. The Task 2 (Fountain) was over before it could begin, since the participant recalled the name of the fountain and even how old it is from memory. The Task 3 (Supervisor) was harder for him – at the beginning, he was searching for the supervisor's name on the screen. After some time, he decided to click on the Help button in the HUD. He started to read and eventually reached the image with the Menu button and its commentary that information about the diploma thesis can be found there. He then proceeded with the rest of the task with no problems. The Task 4 (Rooftops) was unsuccessful – he searched the area for stairs and did know what to do otherwise. He was surprised when he was told about the possibility of flying – he simply did not expect that it is possible to fly, even in the virtual world. When he was searching for the stairs, he even opened the list of Viewpoints from the HUD, but he overlooked the viewpoint on the roof a instead proceeded to click on the Stag Moat viewpoint. He was a little surprised that he was able to teleport there. In a short discussion after the session, the participant said that he liked the experience a lot and did not know that something like virtual walks are even possible and thus was a little shocked about the possibilities. However,

[1]) We have added captions above all the HUD buttons based on the testing done by the Participant 1. More on this in the next section containing the conclusion.

he would have welcomed if the interesting objects were marked better, because he did not know which objects are interactive and which not.

11.2.3 Conclusion

The application was received well by the users who participated in the testing. We have found that even the users with little to none experience regarding 3D and games are able to control the main parts of the application. It has been shown that HUD controls, especially the virtual keyboard, are very suitable for them and they like it. It has proved to be a good idea to include various means of navigation within the virtual world, aimed at both less experienced and more experienced users. The less experienced users are able to use the Viewpoint list, and slightly more experienced are even able to use the Big map. As was expected, very experienced users use exclusively keyboard for the navigation and welcome more advanced features of the application and possibility to change various settings. The user interface (mainly the HUD) got a very positive response – the users appreciated its simplicity and arrangement. Inclusion of the Help button directly into the HUD also proved to be a good idea, because users noticed it quickly and used for guidance when necessary. As for the visual quality, they praised the appearance of the application and were satisfied with the virtual world. On the other side, they would welcome more content, which is something we unfortunately cannot easily add due to the time required for such task and must leave it for the future improvements. One user also noticed that some models are better than others, which is true, but unfortunately it is not something easily fixed. You can see screenshots of the final application in the Appendix C.

The testing also shown us some problems in our design and gave us ideas for improvements. We have made the following changes:

- we have added captions above the HUD buttons
  - so the users do not have to wonder what the button is for
  - we have added a black outline to the captions of all HUD buttons, so they are visible even when the image is very bright (i.e. when looking into the sun)
- we have added HUD buttons for controlling the flying altitude of the avatar
  - it was difficult to control the avatar for the users who were not versed in 3D and games
  - the buttons are visible only when the FLY mode is active
- we have added a HUD button for highlighting objects of interest in the scene
  - it was shown that some users would prefer if objects of interest stand out more
  - entire objects are highlighted in a color
- we made the Minimap clickable
  - users tried to click on the Minimap and were surprised that nothing happened
  - clicking on the Minimap now opens the Bigmap

Upon the analysis of the users' opinions, we have also thought about some possible future improvements to the virtual walk functionality:

- HUD buttons for looking up and down
  - some less experienced users did not know about the mouse look

- Tutorial
  - to clearly state what is possible in the virtual world and how to achieve that, short interactive tutorial explicitly showing the possibilities – e.g. video-like tutorial with a brief text commentary about what is happening on the screen, sometimes requesting the user cooperation to make the controls familiar to him
- Virtual Guided Tour
  - avatar would be walking automatically on a pre-defined tour – the user would not have to worry about controlling the application if he/she does not feel like it

11.3 Performance testing

The application was tested on 2 different machines (see Table 11.2) and 3 major web browsers (see Table 11.3). The tests were conducted three times to eliminate deviations, the results were averaged. Internal Three.js benchmark (Stats.js) was used to measure the performance. The testing scenario consisted of walking around the castle area.

Item / Name	Desktop 2007	Laptop 2009
Manufacturer Name	—	Acer Aspire 3810TZG
Type	Desktop PC	Laptop PC
Year	2007	2009
Processor	Intel Core2 Duo E6850 3.00 GHz	Intel SU4100 1.3 GHz
RAM	4GB	4GB
Graphics card	ATI Radeon HD 3870 512MB	ATI Mobility Radeon HD 4330 512MB Intel GMA 4500MHD
Resolution	1680×1050	1366×768
OS	Win 7 64-bit	Win 7 64-bit

Table 11.2. Hardware and software specifications of machines used for the performance testing

Browser	Version
Mozilla Firefox	43
Google Chrome	46
Opera	33

Table 11.3. Software specifications of web browsers used for the performance testing

Option	High	Medium	Low
Resolution	Full	Full	Half
SSAO	on	off	off
FXAA	on	off	off
Lens Flare	on	off	off
Shadows	on	on	off

Table 11.4. Performance test – description

Browser	High	Medium	Low
Mozilla Firefox	60	60	60
Google Chrome	60	60	60
Opera	60	60	60

Table 11.5. Desktop 2007 – measured FPS

Browser	High	Medium	Low
Mozilla Firefox	17	30	50
Google Chrome	18	35	55
Opera	17	34	55

Table 11.6. Laptop 2009 – measured FPS

As we can see on Table 11.5, our desktop testing machine performed very well and the application was running smoothly. More interesting are results from Table 11.6, where our application had to run on weaker hardware. Running application on laptop with high details was slow – although the laptop has dedicated graphics card, it is simply not strong enough. However, medium details, which still contain shadows, are running above 30 FPS and the experience is alright. Low details were of course the fastest, with shadows off and resolution cut in half. The absence of shadows helped the framerate to became bigger, but since shadows are not actually updated each frame but only when the sun moves (which is about each 15th frame), the main speedup came from the decreased resolution, which was expected.

11.3.1 Conclusion

We have found out that older desktop computers which we target should not have problems running the application smoothly in high details. However, the situation with laptops is worse: dedicated graphics card is an absolute must for older laptops and running the application in high details is not possible. On the other side, medium details can be handled even by older laptops at around 30 FPS. This means we have been successful in our goal of reaching 30 FPS+ on target hardware.

Chapter 12
Conclusion

The aim of this thesis was to develop a web-based 3D virtual walk application visualizing the Prague Castle, provide an insight into the area of virtual walk applications in the context of present-day technologies and explain how to achieve quality visual presentation while retaining high framerate.

Similar visualization systems have been analyzed, which served as the main source of the inspiration during formulation of requirements for the application. We have found out that popular technologies used for realization of web-based virtual walk applications in the past (VRML) and even some promising nowadays ones are not an option, either due to the ending support of external plugins in web browsers or lack of support for a technology in browsers. Therefore, due to the supportability (and performance) reasons, the graphics is rendered using WebGL (with Three.js as the middleware above) and the application itself is implemented in JavaScript.

Realization of the user interface which fulfills identified requirements of the users has been discussed. The application can be controlled by the keyboard, mouse or from HUD. It is localized into Czech and English, with an option of switching between at the runtime. The users can choose to visit interesting places using a graphic list of viewpoints, also they can interact with interesting objects in the virtual world and educate themselves by clicking on them; they have a both classic map and permanently visible small version of classic map available as well, to help them with the orientation in the virtual world. Although the reactions of the users during user testing were positive, a few issues were revealed; some of them were corrected and some were left for the future work.

We have explained our approach taken in the process of virtual content creation and presented the reader with our approach in a form of a guide covering all phases of the process (physical data acquisition, modeling, creating textures, importing into the application). Part of the Prague Castle has been created, along with the surrounding 12 km^2 of the terrain generated from the real digital elevation model of Prague.

Collision detection is handled by our own simple heightmap-based approach which allows us to have even overhang areas to some degree, which is not usually possible with heightmaps. We have employed performance optimizations which focus on reducing WebGL state changes by utilizing technique of Mesh merging. Further performance gain was obtained by reduction of drawing calls, which was possible thanks to the rearrangement of meshes' data where triangles sharing same material are stored in contiguous blocks of memory. We have incorporated algorithms improving visual quality such as Fast Approximate Anti Aliasing for fixing aliasing artifacts, global illumination technique of Screen Space Ambient Occlusion or shadows via Shadow Mapping and still maintained smooth framerate on the target hardware. Visual effects include also a day/night cycle and another minor effects such as lens flare or bump/normal mapping. The application runs in majority of web browsers (Chrome, Firefox, Opera) and requires no external plugin.

12.1 Future work

The future work can be divided into the three categories. The first category is a **virtual content creation**. Current application contains the top part of the Prague Castle where ceremonials are held, but the central part with the Third Courtyard and the bottom part containing e.g. the famous Golden Lane are yet to be done. Creating these areas will be a very time demanding task. The second category of the future work is solving minor issues related to UI which were revealed by the user testing and also **enhancing the virtual walk functionality** even further, for example by virtual guided tours. The third category are **additional visual quality and performance improvements**. The application is fast enough to handle over a hundred of thousands of triangles and hundreds of different materials (depending on the quality settings), but there is still room for an improvement – the user interface is already very mobile-friendly, so reducing hardware requirements even further is the last remaining step for making it possible to run the application at smooth speed even on mobile phones or tablets.

References

[1] Wikipedia. Google Earth.
http://en.wikipedia.org/wiki/Google_Earth [cit. 09-30-2014].

[2] Google Inc. Google Earth downloaded more than one billion times. Google Official Blog.
http://googleblog.blogspot.cz/2011/10/google-earth-downloaded-more-than-one.html [cit. 09-30-2014].

[3] Google Inc. User generated 3D model pipeline has been retired October 1st, 2013. Google Groups.
https://groups.google.com/forum/#!msg/3dwh/epXUQA2bJ2s/pw7G8E6wtZ4J [cit. 09-30-2014].

[4] Google Inc. What happened to Google Earth 3D view for Los Angeles. Google Groups.
https://productforums.google.com/forum/#!msg/gec-open-forum/lv_K22404q4/2Knak2Sy2qEJ [cit. 09-30-2014].

[5] Google Inc. The Next Dimension of Google Maps - Official Post. Google Groups.
https://productforums.google.com/forum/#!msg/gec-open-forum/lv_K22404q4/2Knak2Sy2qEJ [cit. 09-30-2014].

[6] gisportal.cz. Nejrozsáhlejší 3D mapa na světě vzniká českou technologií Melown Maps.
http://www.gisportal.cz/2014/04/nejrozsahlejsi-3d-mapa-na-svete-vznika-ceskou-technologii-me
[cit. 10-02-2014].

[7] Marek Dobrý. Přichází nové Mapy.cz. Neztratíte se ani na webu, ani v ulicích. Seznam.cz věstník, March 2014.
http://newsletter.seznam.cz/articles/120 [cit. 10-02-2014].

[8] Pino Bonetti. Ovi Maps 3D: The World is Not Flat.
http://360.here.com/2011/04/19/ovi-maps-3d-the-world-is-not-flat [cit. 10-04-2014].

[9] VOP Project Group. Documentation – Introduction.
http://dcgi.felk.cvut.cz/cgg/vsp2/about/1.html [cit. 10-06-2014].

[10] User with nickname "patapom". Virtual 3D visit, the Saint Jean Cathedral.
http://patapom.com/topics/WebGL/cathedral/intro.html [cit. 10-08-2014].

[11] TimeWalk: Your Virtual Reality Playground.
https://www.indiegogo.com/projects/timewalk-your-virtual-reality-playground#gallery
[cit. 10-08-2014].

[12] Benjamin Smedberg, Mozilla. NPAPI Plugins in Firefox.
https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox
[cit. 11-18-2015].

[13] Justin Schuh, Google Inc. The Final Countdown for NPAPI.
http://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html [cit. 11-18-2015].

[14] Web3D Consortium. What is X3D.
http://www.web3d.org/what-x3d [cit. 10-10-2014].

[15] x3dom.org. About X3D.
http://www.x3dom.org/?page_id=2 [cit. 10-11-2014].

[16] Danny Winokur. Flash to Focus on PC Browsing and Mobile Apps.
http://blogs.adobe.com/conversations/2011/11/flash-focus.html [cit. 10-12-2014].

[17] Marco Scabia. How Stage3D works.
http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html [cit. 10-12-2014].

[18] Google Inc. Technical Overview.
http://developer.chrome.com/native-client/overview [cit. 10-13-2014].

[19] Cade Metz. Mozilla: Our browser will not run native code.
http://www.theregister.co.uk/2010/06/24/jay_sullivan_on_firefox [cit. 10-13-2014].

[20] Cade Metz. Google Native Client: The web of the future - or the past?.
http://www.theregister.co.uk/2010/06/24/jay_sullivan_on_firefox [cit. 10-14-2014].

[21] Wikipedia. Source-to-source compiler.
http://en.wikipedia.org/wiki/Source-to-source_compiler [cit. 10-14-2014].

[22] Writing Definition (.d.ts) Files.
http://typescript.codeplex.com/wikipage?title=Writing%20Definition%20%28.d.ts%29%20Files [cit. 10-15-2014].

[23] Dart VM and dart2js Performance.
http://www.dartlang.org/performance [cit. 10-15-2014].

[24] Lars Bak and Kasper Lund, Google Inc. Dart for the Entire Web .
http://news.dartlang.org/2015/03/dart-for-entire-web.html [cit. 11-21-2015].

[25] Brendan Eich
http://news.ycombinator.com/item?id=2982949 [cit. 10-16-2014].

[26] Emscripten documentation.
http://kripken.github.io/emscripten-site [cit. 10-16-2014].

[27] David Herman, Luke Wagner, Alon Zakai. Specification (working draft).
http://asmjs.org/spec/latest [cit. 10-17-2014].

[28] Alon Zakai, Robert Nyman. asm.js performance improvements in the latest version of Firefox make games fly!.
http://hacks.mozilla.org/2014/05/asm-js-performance-improvements-in-the-latest-version-of-fi
[cit. 10-17-2014].

[29] Mozilla. Mozilla and Epic Preview Unreal Engine 4 Running in Firefox.
http://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-
[cit. 10-17-2014].

[30] Fraunhofer IGD/VCST. FAQ — X3DOM Documentation.
http://x3dom.readthedocs.org/en/1.4.0/notes/faq.html [cit. 12-7-2014].

[31] stackoverflow.com SceneJS vs Three.JS vs others.
`http://stackoverflow.com/a/6965426` [cit. 12-31-2014].

[32] Alain Abran. Guide to the software engineering body of knowledge 2004 version: SWEBOK. Los Alamitos, Calif, IEEE Computer Society Press, 2004.

[33] Zara Jiri, Chromy Pavel, Cizek Jiri, Ghais Kamil, Holub Michal, Mikes Stanislav, Rajnoch Jakub. A Scaleable Approach to Visualization of Large Virtual Cities. Proceedings Fifth International Conference on Information Visualisation, Los Alamitos, IEEE Computer Society Press, p. 639-644, 2001.

[34] W3Schools. Browser Statistics.
`http://www.w3schools.com/browsers/browsers_stats.asp` [cit. 11-20-2015].

[35] Statista.com Global forecast of the average PC age from 2006 to 2015 (in years).
`http://www.statista.com/statistics/203817/global-forecast-of-the-average-age-of-pcs-up-to-20`
[cit. 11-20-2015].

[36] Jason Gregory. Game Engine Architecture. CRC Press, 2009.

[37] Microsoft Corporation. Key Principles of Software Architecture. Microsoft Developer Network.
`https://msdn.microsoft.com/en-us/library/ee658124.aspx` [cit. 11-18-2015].

[38] webglstats.com WebGL Stats.
`http://webglstats.com` [cit. 11-7-2015].

[39] Mozilla. Using textures in WebGL.
`https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Using_textures_in_WebGL` [cit. 12-15-2015].

[40] GIMP Plugin Registry Make Seamless Advanced.
`http://registry.gimp.org/node/28112` [cit. 11-07-2015].

[41] Geoportal Praha. Shuttle Radar Topography Mission.
`http://www.geoportalpraha.cz/` [cit. 11-07-2015].

[42] Fang Qiu. Synergy of LIDAR and High-Resolution Digital Orthophotos to Support Urban Feature Extraction and 3d City Model Construction. Geospatial Information Sciences, The University of Texas at Dallas.
`http://www.utsa.edu/lrsg/Teaching/EES5053-06/Qiu_UTD_Lidar.pdf` [cit. 11-07-2015].

[43] Wikipedia. Surfaces represented by a Digital Surface Model and Digital Terrain Model.
`https://en.wikipedia.org/wiki/File:DTM_DSM.svg` [cit. 11-07-2015].

[44] Prague Institute of Planning and Development. Prague: Digital terrain model (raster).
`http://opendata.iprpraha.cz/CUR/D3M/DTM1M/DTM1M.tif` [cit. 11-07-2015].

[45] Seznam.cz, OpenStreetMap, NASA. Geographical map of Prague.
`http://mapy.cz/zemepisna?x=14.4314998&y=50.0593852&z=11` [cit. 11-07-2015].

[46] Žára Jiří. Web-Based Historical City Walks: Advances and Bottlenecks. PRESENCE: Teleoperators and Virtual Environments, 2006, vol. 15, no. 3, p. 262-277.

[47] caniuse.com SVG (basic support): Method of displaying basic Vector Graphics features using the embed or object elements.
`http://caniuse.com/#feat=svg` [cit. 11-08-2015].

[48] Kouichi Matsuda, Rodger Lea. WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL (OpenGL). Addison-Wesley Professional, 2013.

[49] Wikipedia. Newton's law of universal gravitation.
https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation [cit. 11-12-2015].

[50] Prague Institute of Planning and Development. Aerial images (orthophotomaps) – Archive.
http://mpp.praha.eu/OrtofotoArchiv/default.aspx [cit. 12-11-2015].

[51] Skytiger. Packing Depth into Color – New 24 bit Packing.
https://skytiger.wordpress.com/2010/12/01/packing-depth-into-color/ [cit. 12-11-2015].

[52] Wikipedia. Free fall.
https://en.wikipedia.org/wiki/Free_fall [cit. 11-14-2015].

[53] Opengl.org S3 Texture Compression.
https://www.opengl.org/wiki/S3_Texture_Compression [cit. 12-01-2015].

[54] Macey Jon. Illumination models.
https://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/IlluminationModels4up.pdf [cit. 11-13-2015].

[55] Agu Emmanuel. llumination and Shading.
http://web.cs.wpi.edu/~emmanuel/courses/cs543/slides/lecture05_p2.pdf [cit. 11-13-2015].

[56] Barré-Brisebois, Colin. Finding Next-Gen – Part I – The Need For Robust (and Fast) Global Illumination in Games.
http://madebyevan.com/webgl-path-tracing/ [cit. 11-14-2015].

[57] Tabellion Eric. Global Illumination Across Industries, SIGGRAPH 2010 Course.
http://cgg.mff.cuni.cz/~jaroslav/gicourse2010/giai2010-04-eric_tabellion-slides.pdf [cit. 11-14-2015].

[58] Boesch Florian. WebGL Deferred Irradiance Volumes.
http://codeflow.org/entries/2012/aug/25/webgl-deferred-irradiance-volumes/ [cit. 11-14-2015].

[59] Wallace Evan. WebGL Path Tracing.
http://madebyevan.com/webgl-path-tracing/ [cit. 11-14-2015].

[60] Cutler Barb. Local vs. Global Illumination and Radiosity.
http://www.cs.rpi.edu/~cutler/classes/advancedgraphics/F05/lectures/14_radiosity.pdf [cit. 11-13-2015].

[61] Asawari Deshpande, Siddhartha Singh Sandhu, Shruti Gotmare, Vineet Mahadik. Department of Computer Science Ambient Light Occlusion & Shadows using WebGL. Department of Computer Science, University of Southern California, Los Angeles, USA.

[62] Kvarfordt Daniel, Lillandt Benjamin. Screen Space Ambient Occlusion.
http://www.cse.chalmers.se/edu/course/TDA361/Advanced%20Computer%20Graphics/SSAO.pdf [cit. 11-17-2015].

[63] Bittner Jiří. Shadows in rasterization pipeline. Algorithms of Computer Graphics – Lecture 11. Czech Technical University in Prague.

[64] Ambrož David, Felkel Petr. Shadows for real-time graphics. Computer Graphics 2 – Lecture 9. Czech Technical University in Prague.
https://cent.felk.cvut.cz/courses/PGR2/lectures/09-gpu_shadows.pdf [cit. 12-04-2015].

[65] Scott Todd. Shadow volumes visualized. GitHub.
http://scotttodd.github.io/assets/projects/graphics/shadow-volumes.png [cit. 12-04-2015].

[66] Joshua Koo. webgl - sky + sun shader. Three.js examples.
http://threejs.org/examples/#webgl_shaders_sky [cit. 12-31-2015].

[67] Jerome Etienne, John Chapman. Pseudo Lens Flare.
https://github.com/jeromeetienne/threex.sslensflare [cit. 12-31-2015].

[68] dahlsys.com Anti-aliasing technologies.
http://www.dahlsys.com/misc/antialias [cit. 11-17-2015].

[69] Warner Mark. NVIDIA's New FXAA Antialiasing Technology.
http://www.hardocp.com/article/2011/07/18/nvidias_new_fxaa_antialiasing_technology/5#.VnVuH7_ypvU [cit. 11-18-2015].

[70] DesLauriers Matt. three-shader-fxaa. GitHub.
https://github.com/mattdesl/three-shader-fxaa [cit. 11-18-2015].

[71] User Experience Professionals' Association. Think Aloud Testing. The Usability Body of Knowledge.
http://www.usabilitybok.org/think-aloud-testing [cit. 12-23-2015].

[72] Jason Gregory. Runtime game engine architecture.
http://www.gameenginebook.com/img/fig-runtime-arch.jpg [cit. 11-18-2015].

Appendix A
Abbreviations

AA	Anti-Aliasing
AABB	Axis Aligned Bounding Box
API	Application Interface
BRDF	Bidirectional Reflectance Distribution Function
CVUT	Czech Technical University in Prague
DEM	Digital Elevation Model
DSM	Digital Surface Model
DTM	Digital Terrain Model
DOM	Document Object model
ECMA	European Computer Manufacturers Association
FEE	Faculty of Electrical Engineering
FPS	Frames Per Second
FXAA	Fast Approximation Anti-Aliasing
GPU	Graphics Processing Unit
GWT	Google Web Toolkit
HDD	Hard Disk Drive
HTML	Hyper Text Markup Language
HUD	Head Up Display
IDE	Integrated Development Editor
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
LLVM	Low Level Virtual Machine (Compiler infrastructure designed as a set of reusable libraries with well-defined interfaces)
LOD	Level of Detail
MLAA	Morphological Anti-Aliasing
MSAA	Multi Sample Anti-Aliasing
OS	Operating System
PCF	Percentage Closer Filtering
PHP	Hypertext Preprocessor
PNG	Portable Network Graphics
SRTM	The Shuttle Radar Topography Mission
SSAA	Super-Sampling Anti-Aliasing
SSAO	Screen Space Ambient Occlusion
SDK	Software Development Kit
SQL	Structured Query Language
SVG	Scalable Vector Graphics
UI	User Interface
VM	Virtual Machine
VOP	Virtual Old Prague

Appendix B
Additional images

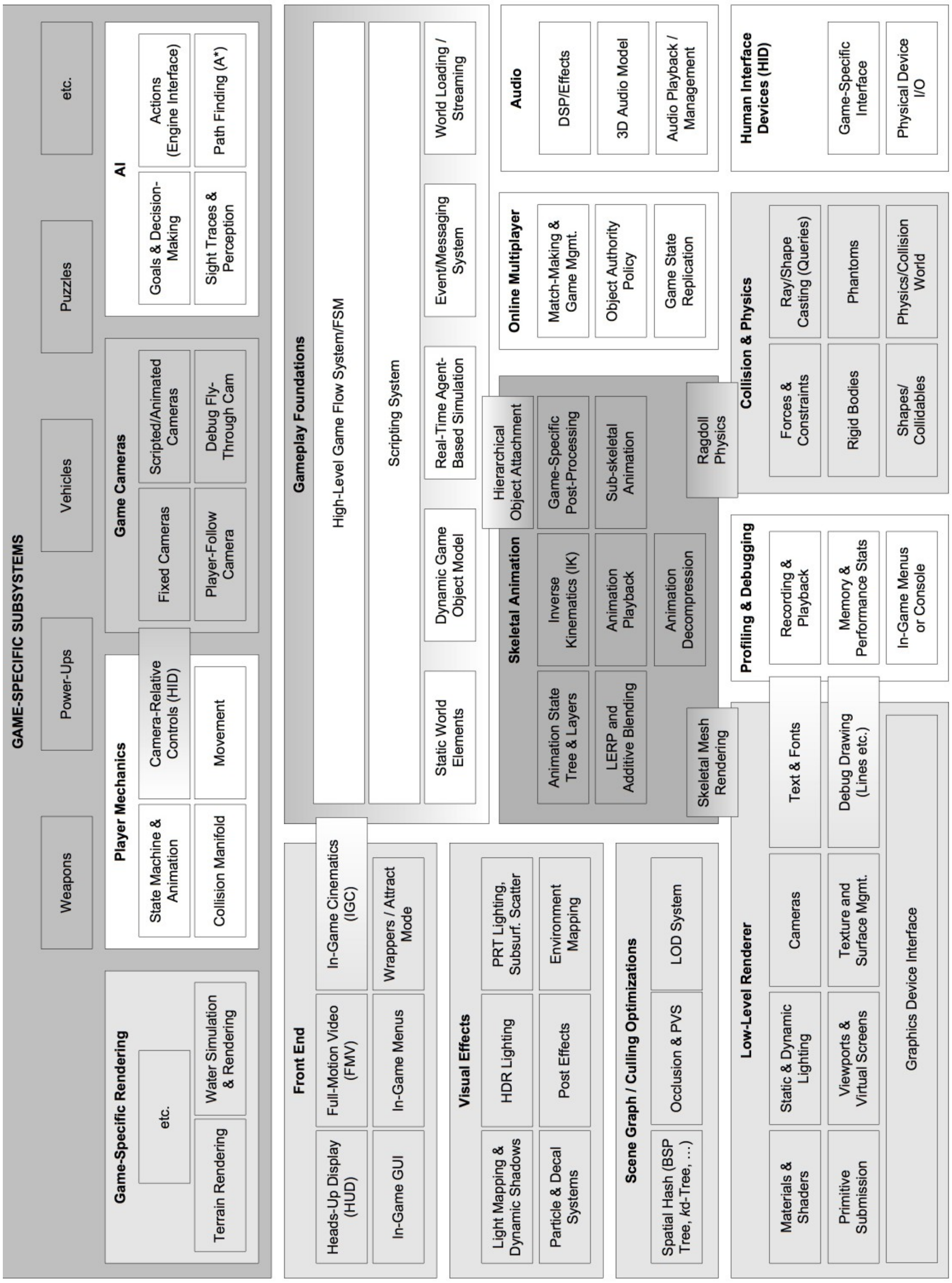


Figure B.1. Runtime game engine architecture part 1/2 [72]

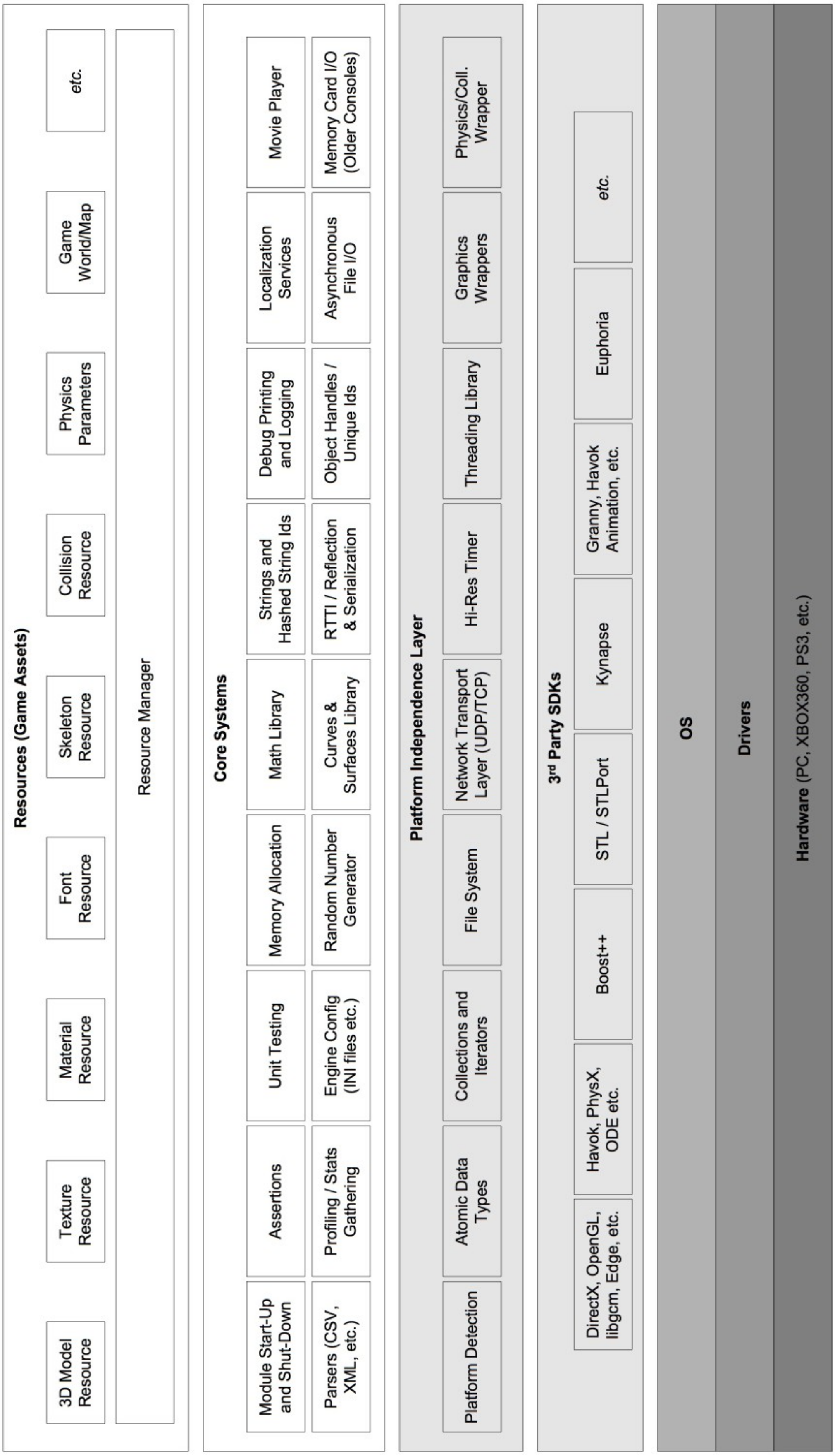


Figure B.2. Runtime game engine architecture part 2/2 [72]

Appendix C
Screenshots and comparisons

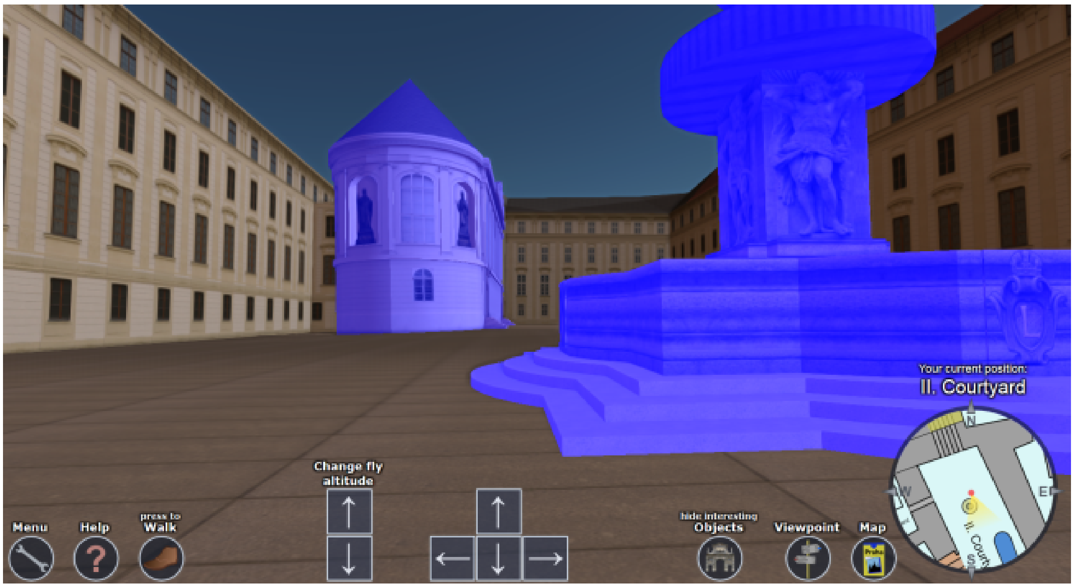


Figure C.3. Color highlighting of interesting objects

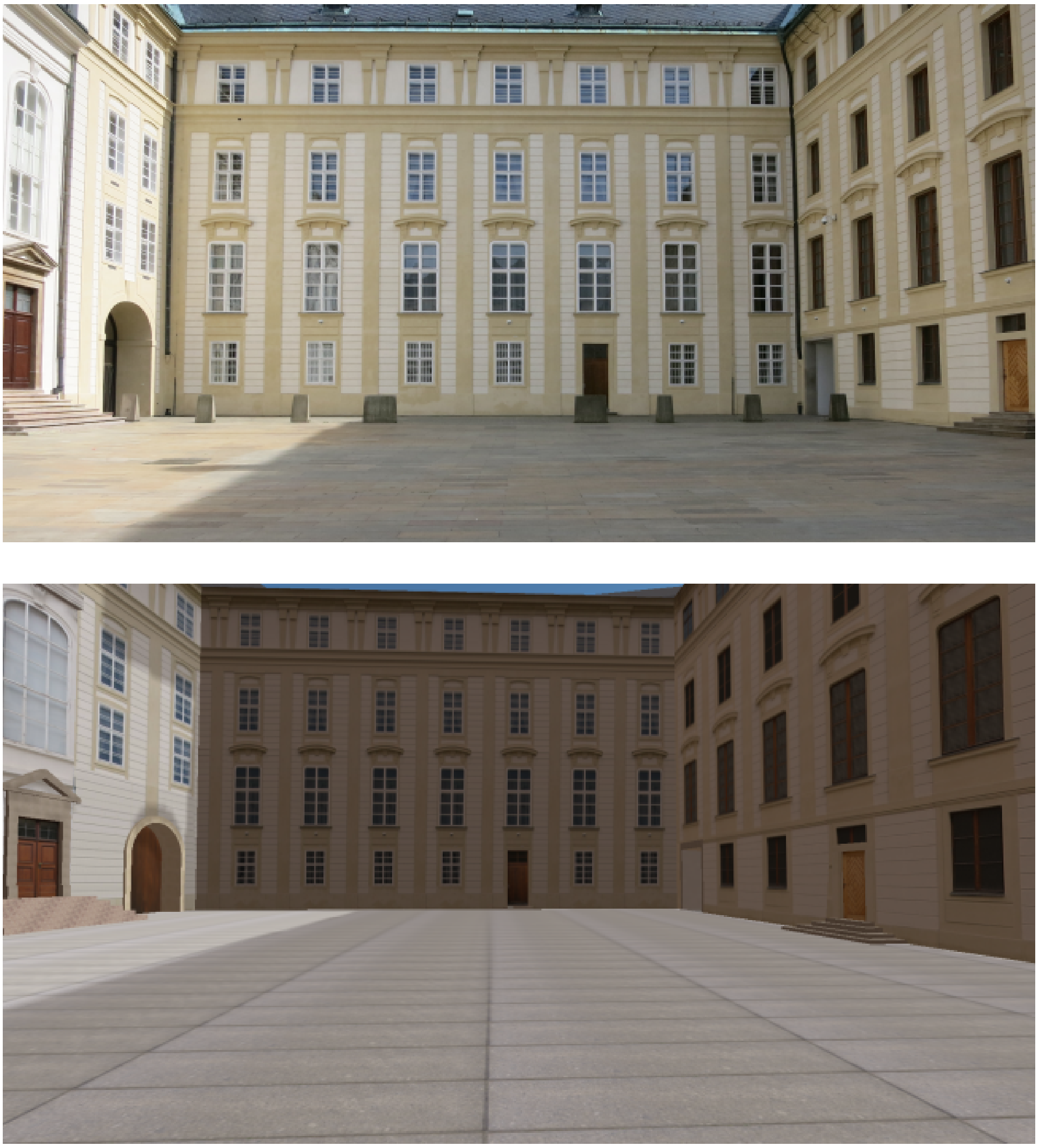


Figure C.4. Reality vs our application – The Second Courtyard

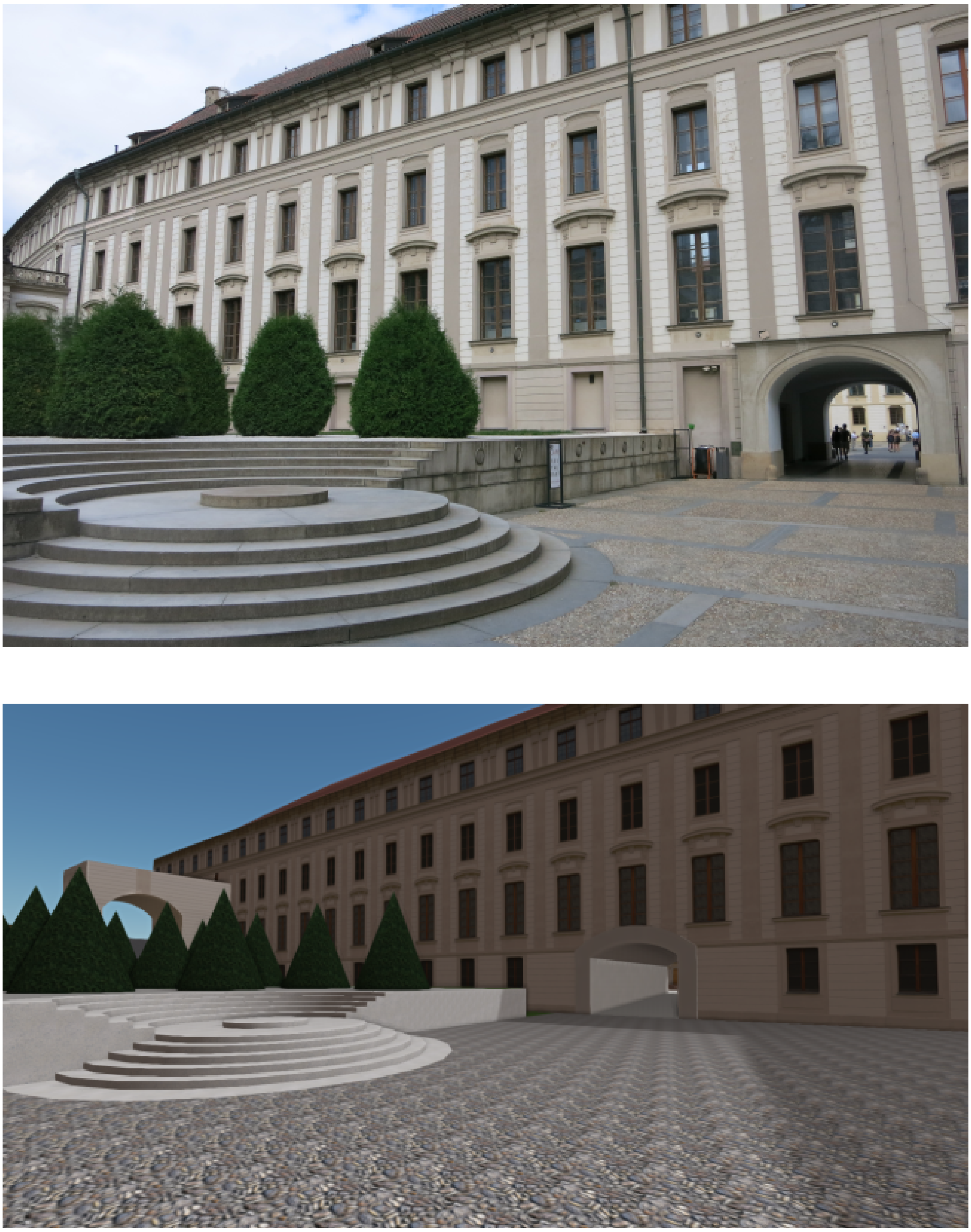


Figure C.5. Reality vs our application – The Fourth Courtyard

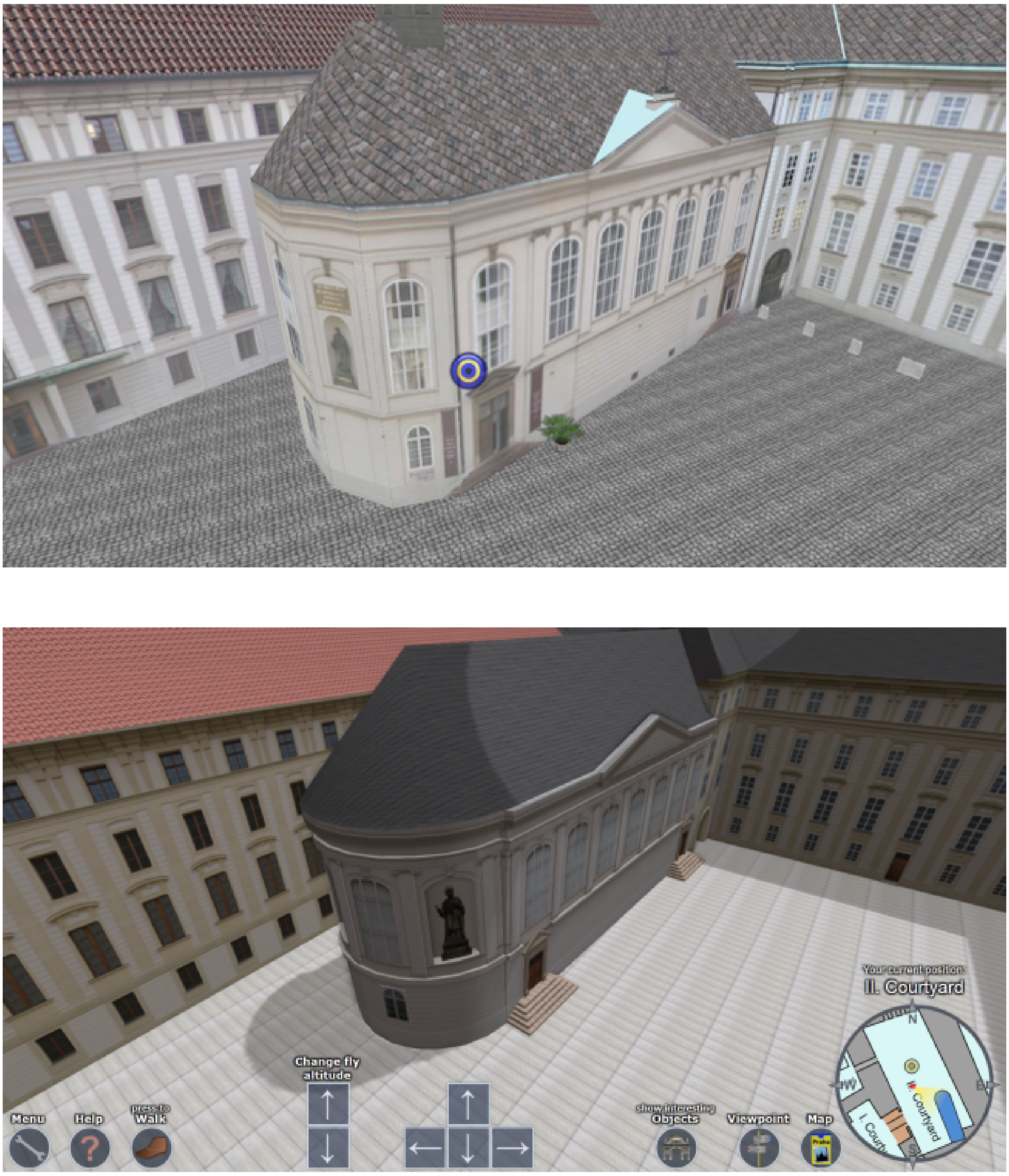


Figure C.6. Virtual Old Prague vs our application – Chapel of the Holy Rood

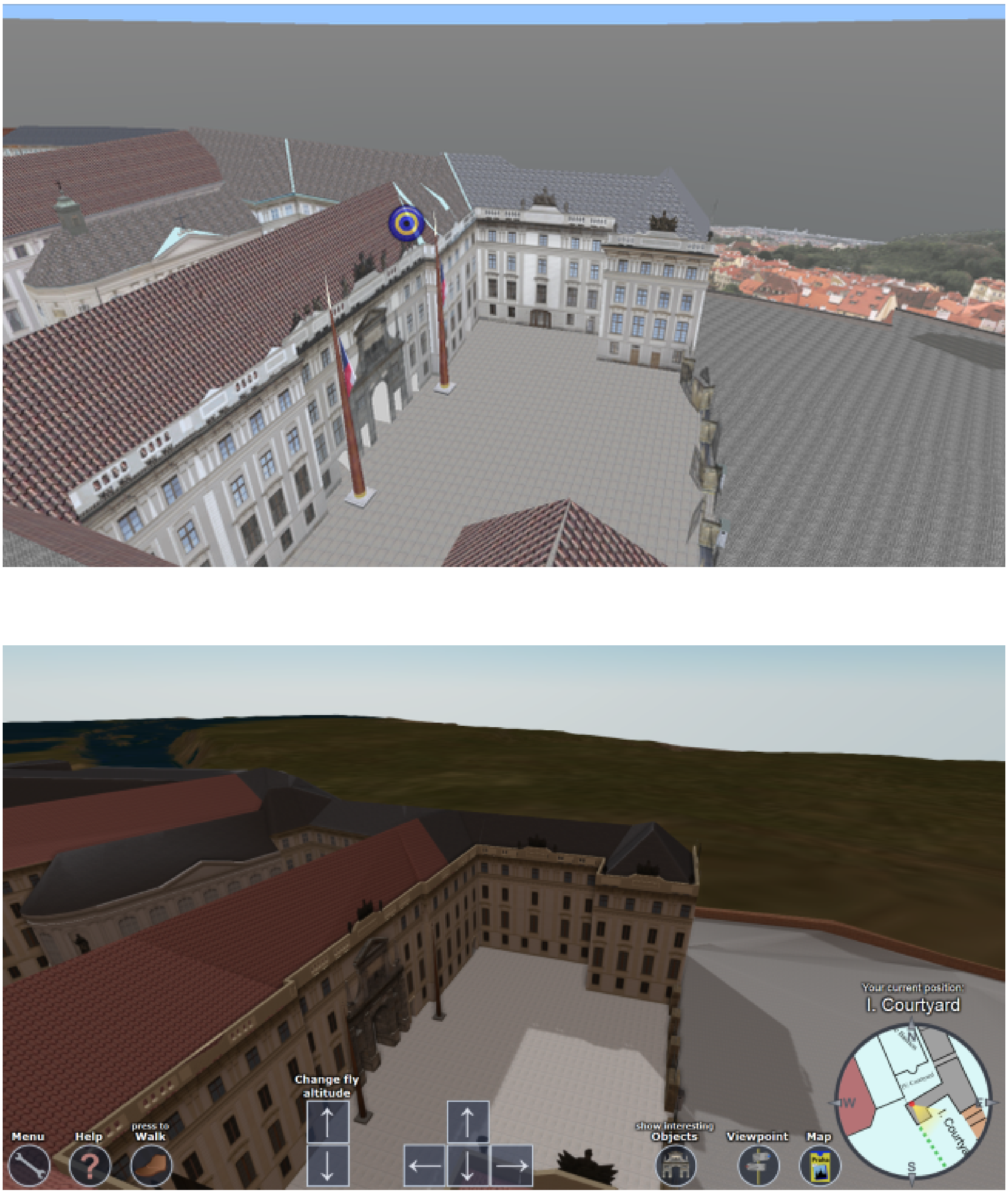
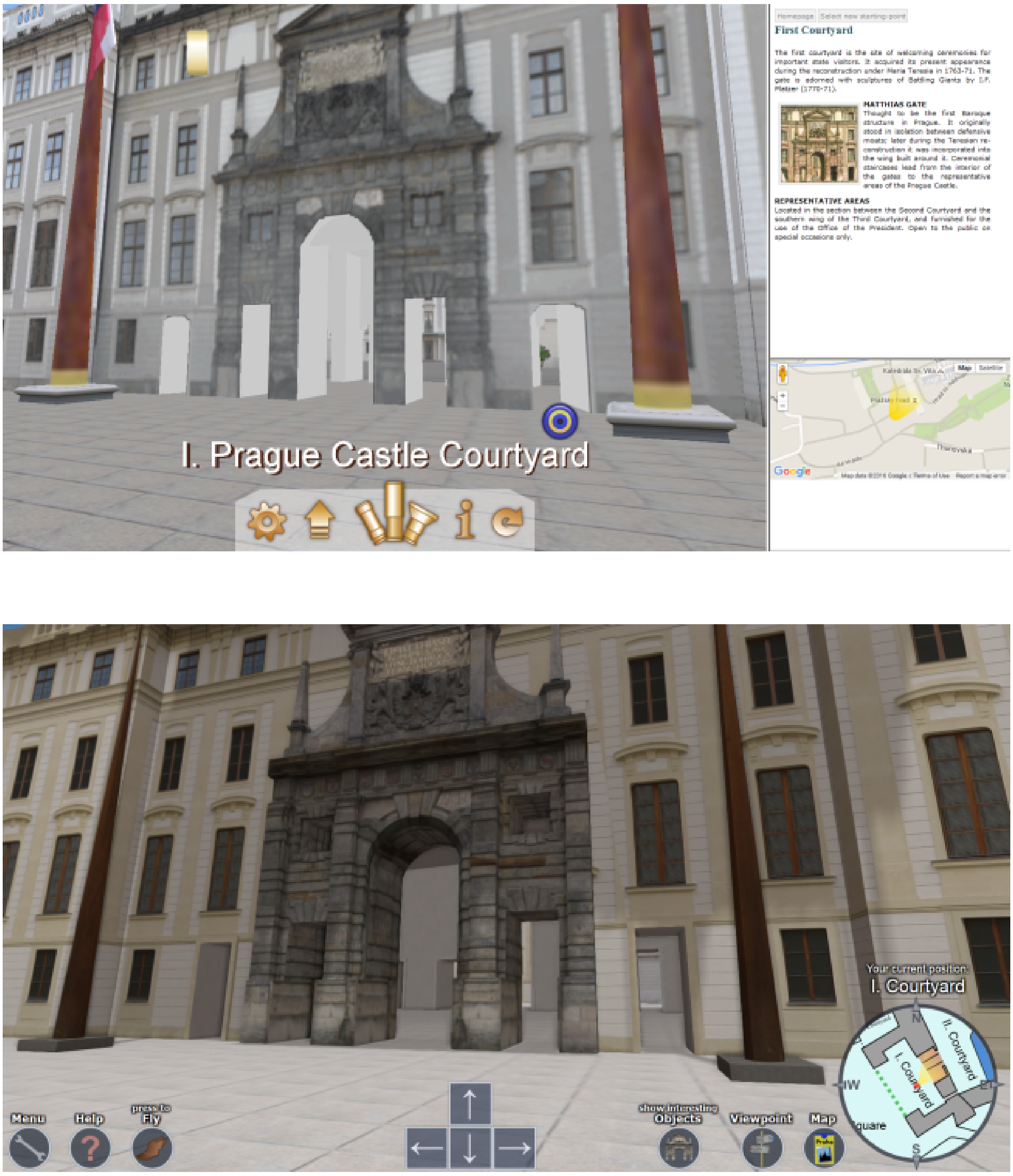


Figure C.7. Virtual Old Prague vs our application – The First Courtyard



First Courtyard

The first courtyard is the site of welcoming ceremonies for important state visitors. It acquired its present appearance during the reconstruction under Maria Teresia in 1763-71. The gate is adorned with sculptures of Battling Giants by I.F. Platzer (1770-71).

MATTHIAS GATE
Thought to be the first Baroque structure in Prague. It originally stood in isolation between defensive moats; later during the Teresian reconstruction it was incorporated into the wing built around it. Ceremonial staircases lead from the interior of the gates to the representative areas of the Prague Castle.

REPRESENTATIVE AREAS
Located in the section between the Second Courtyard and the southern wing of the Third Courtyard, and furnished for the use of the Office of the President. Open to the public on special occasions only.

Figure C.8. Virtual Old Prague vs our application – Matthias Gate

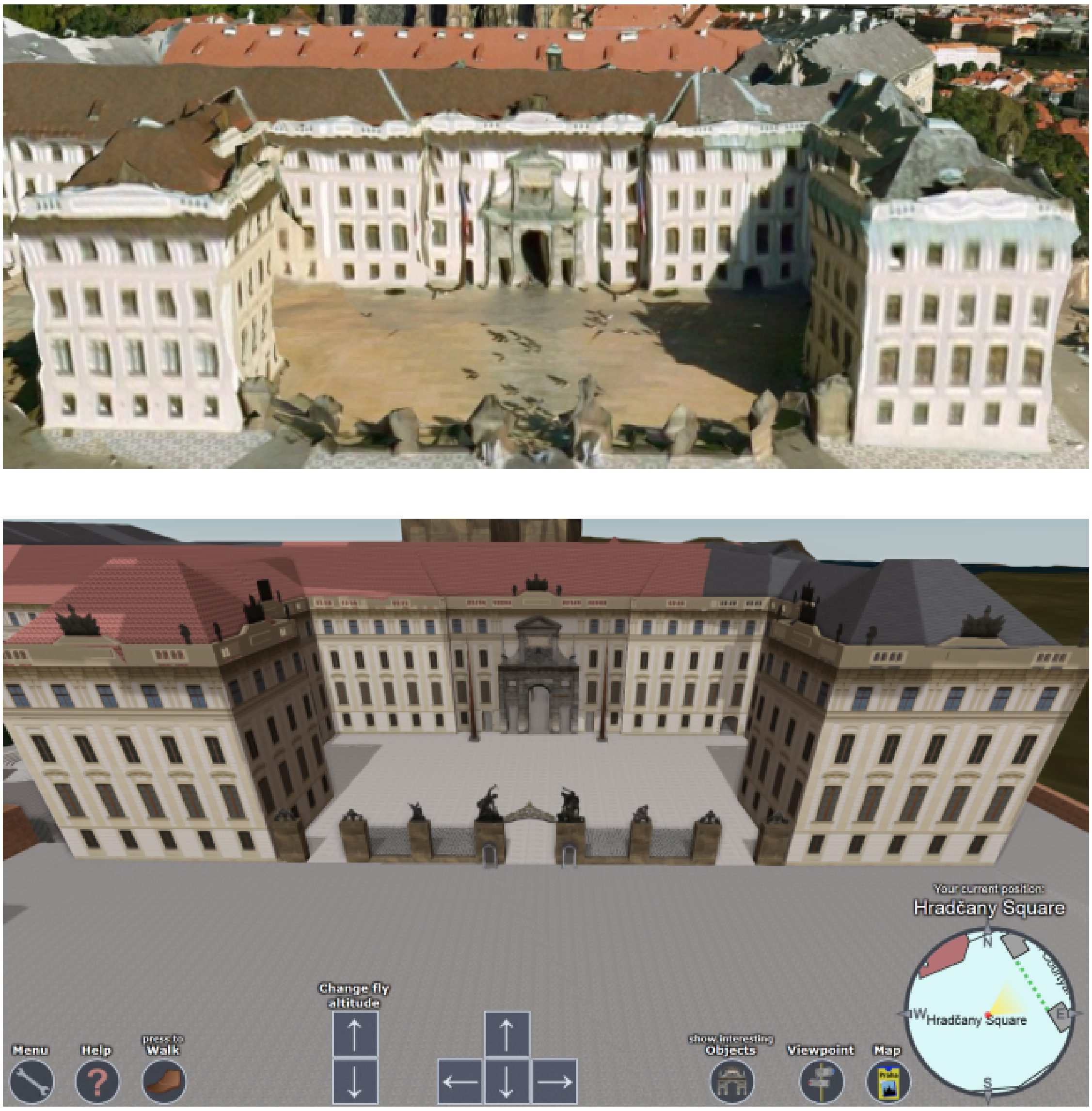


Figure C.9. Mapy.cz vs our application – The First Courtyard

Appendix D
Installation manual

D.1 Running the application

This section covers how to run the application. There are two ways to do that:

- From the web
  - Open the address `http://leyfi.felk.cvut.cz/prague-castle-3d` in your web browser
- Locally
  - Open the file `index.html` or `run_application.html` in a web browser.
  - You may need to adjust your browser's security settings (see the next section D.1.1)

D.1.1 Running the application locally

There is a high probability that because of browsers' security restrictions[1]), loading application files (models, textures, ...) when running the application locally will fail with a security exception. There are two options how overcome the issue:

- Run `index.html` or `run_application.html` from a local server[2])
  - i.e. access the page as `http://localhost/index.html`
- Change browser security policy
  - Mozilla Firefox
    - type `about:config` into the address bar
    - option `security.fileuri.strict_origin_policy` must be `false`
  - Google Chrome, Opera
    - close any running instance of Google Chrome
    - run Chrome from the command line:
      - `chrome --allow-file-access-from-files`

Please note that if you change your browser's security policy, the browser may become vulnerable. We recommend you create a new profile where you make these changes and use it only for running local files and not regular web browsing.

[1]) https://en.wikipedia.org/wiki/Same-origin_policy

[2]) For example Apache – available at https://httpd.apache.org

D.2 Requirements and troubleshooting

Application targets personal computers and laptops. It shall generally run on personal computers manufactured in year 2008 and later, and on laptops manufactured in year 2010 and later.

D.2.1 Minimum computer configuration

- Processor: 1.3 GHz
- RAM: 4 GB
- Graphics card: dedicated with 512MB
- Supported web browser

D.2.2 Supported web browsers

The application supports following web browsers:

- Mozilla Firefox 43+
- Google Chrome 46+
- Opera 33+

Running the application in another browser is not guaranteed to work.

The web browser must have WebGL turned on. To see if WebGL is turned on and works properly, you can visit one of the following sites:

- `https://get.webgl.org`, `http://webglreport.com`

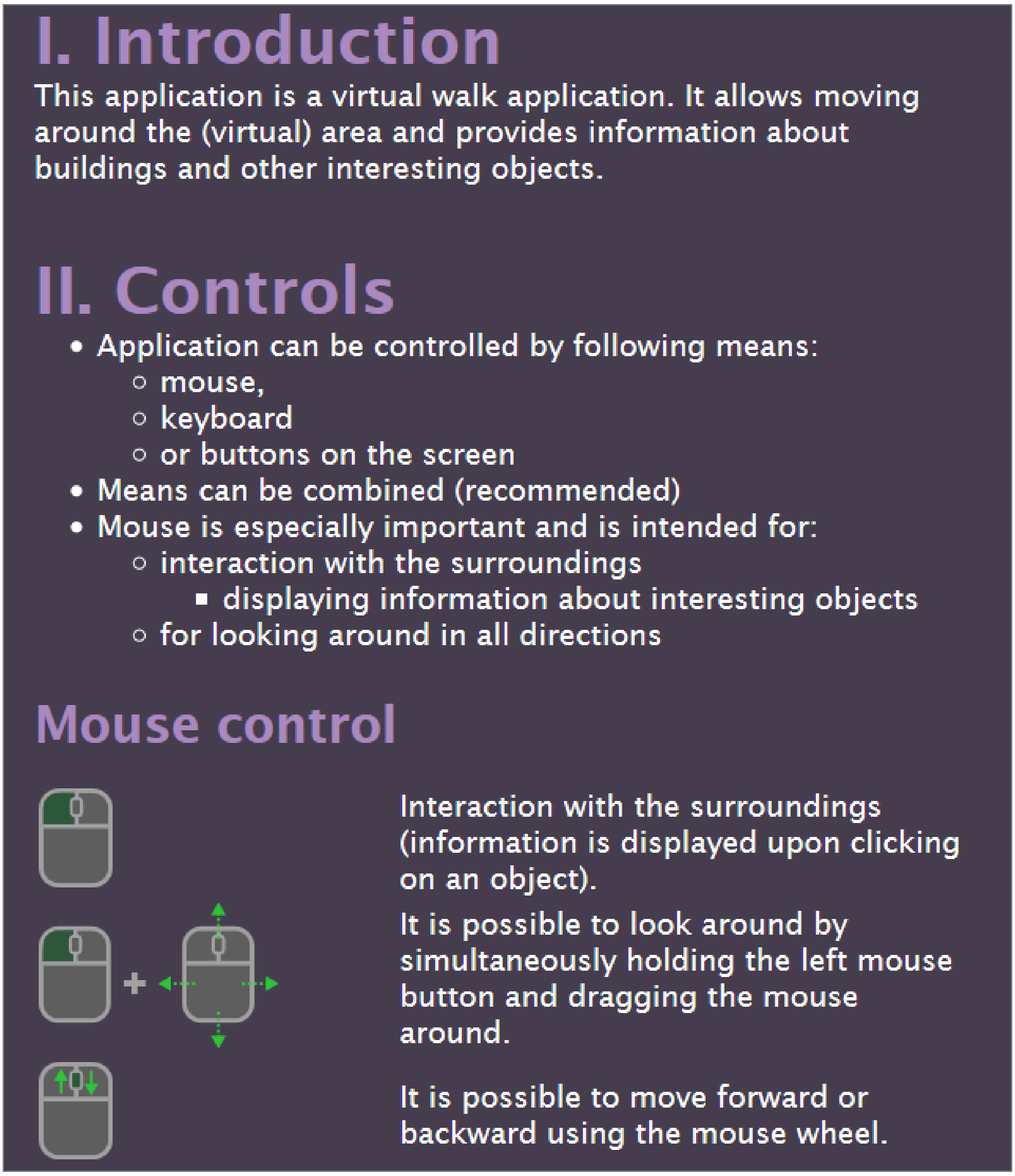
Make sure that hardware acceleration is enabled. It can happen that WebGL is turned on, but hardware acceleration is disabled, making the application run slow. If you have a feeling that the application is slow, we recommend you manually check following options in your browser:

- Mozilla Firefox
    - type `about:config` into the address bar
    - `webgl.disabled` must be `false`
    - `webgl.force-enabled` must be `true`
    - try setting an opposite value for the option `webgl.disable-angle`
- Google Chrome
    - type `chrome://flags` into the address bar
    - `Override software rendering list` must be enabled
    - `Disable WebGL` must be disabled
- Opera
    - type `opera:config` into the address bar
    - `Enable WebGL` and `Enable Hardware Acceleration` must be 2

D.2.3 Graphic card drivers

Make sure your graphics card drivers are up-to-date.

Appendix E
User manual



I. Introduction

This application is a virtual walk application. It allows moving around the (virtual) area and provides information about buildings and other interesting objects.

II. Controls

- Application can be controlled by following means:
  - mouse,
  - keyboard
  - or buttons on the screen
- Means can be combined (recommended)
- Mouse is especially important and is intended for:
  - interaction with the surroundings
    - displaying information about interesting objects
  - for looking around in all directions

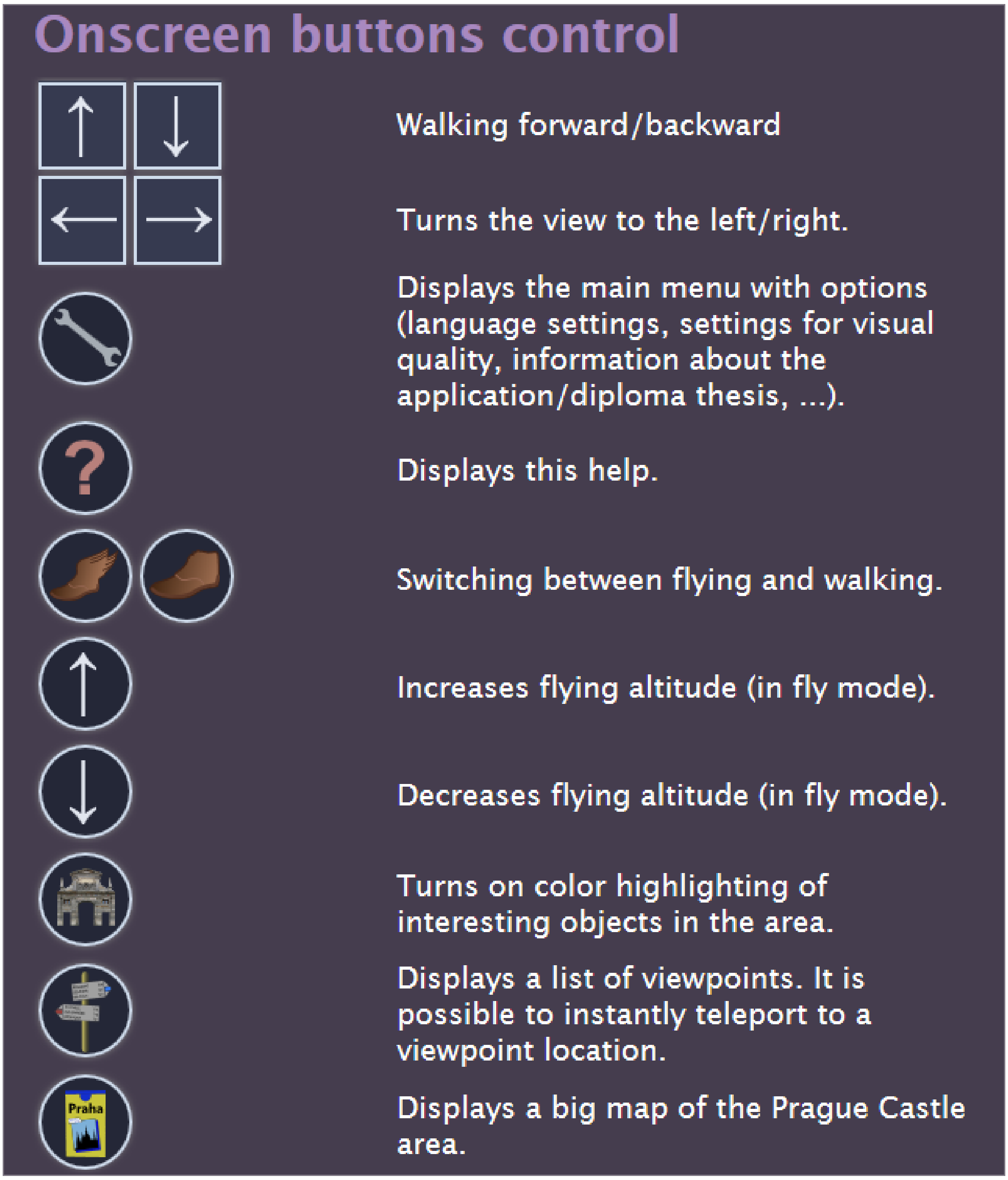
Mouse control

Interaction with the surroundings (information is displayed upon clicking on an object).

It is possible to look around by simultaneously holding the left mouse button and dragging the mouse around.

It is possible to move forward or backward using the mouse wheel.

Figure E.10. User Guide 1/6 – Introduction, Controls



Onscreen buttons control

↑ ↓ Walking forward/backward

← → Turns the view to the left/right.

Displays the main menu with options (language settings, settings for visual quality, information about the application/diploma thesis, ...).

Displays this help.

Switching between flying and walking.

↑ Increases flying altitude (in fly mode).

↓ Decreases flying altitude (in fly mode).

Turns on color highlighting of interesting objects in the area.

Displays a list of viewpoints. It is possible to instantly teleport to a viewpoint location.

Displays a big map of the Prague Castle area.

Figure E.11. User Guide 2/6 – Onscreen buttons control

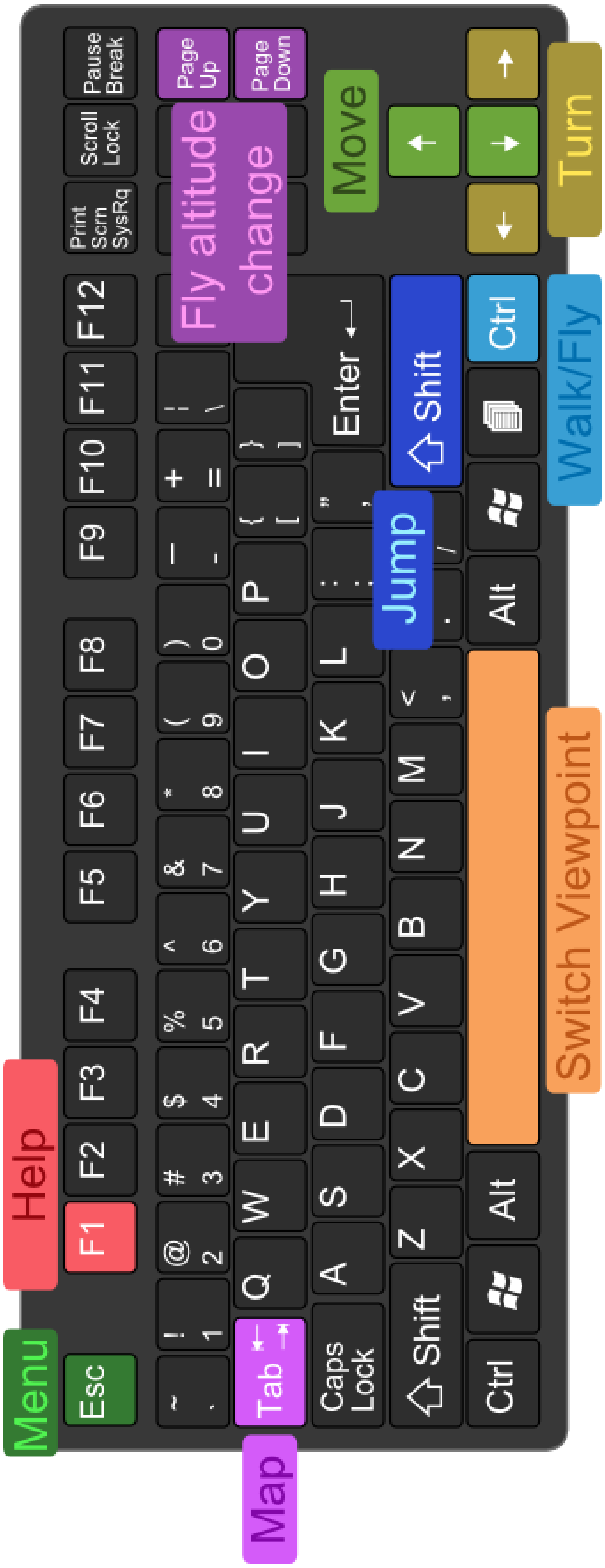
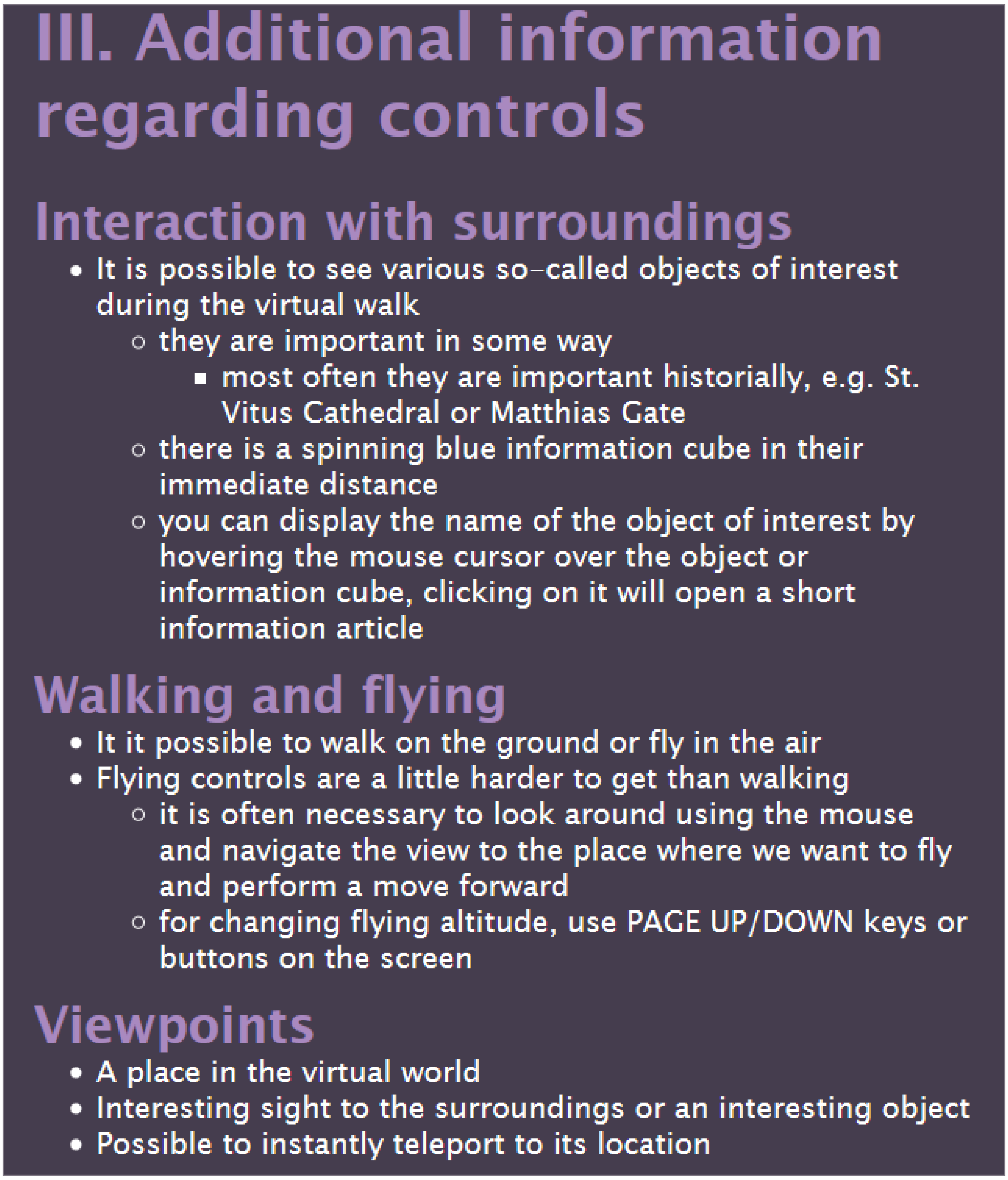


Figure E.12. User Guide 3/6 – Keyboard controls (advanced version; note that the user manual in the application contains also a more basic version and it is possible to switch between the basic and the advanced one)



III. Additional information regarding controls

Interaction with surroundings

- It is possible to see various so–called objects of interest during the virtual walk
  - they are important in some way
    - most often they are important historially, e.g. St. Vitus Cathedral or Matthias Gate
  - there is a spinning blue information cube in their immediate distance
  - you can display the name of the object of interest by hovering the mouse cursor over the object or information cube, clicking on it will open a short information article

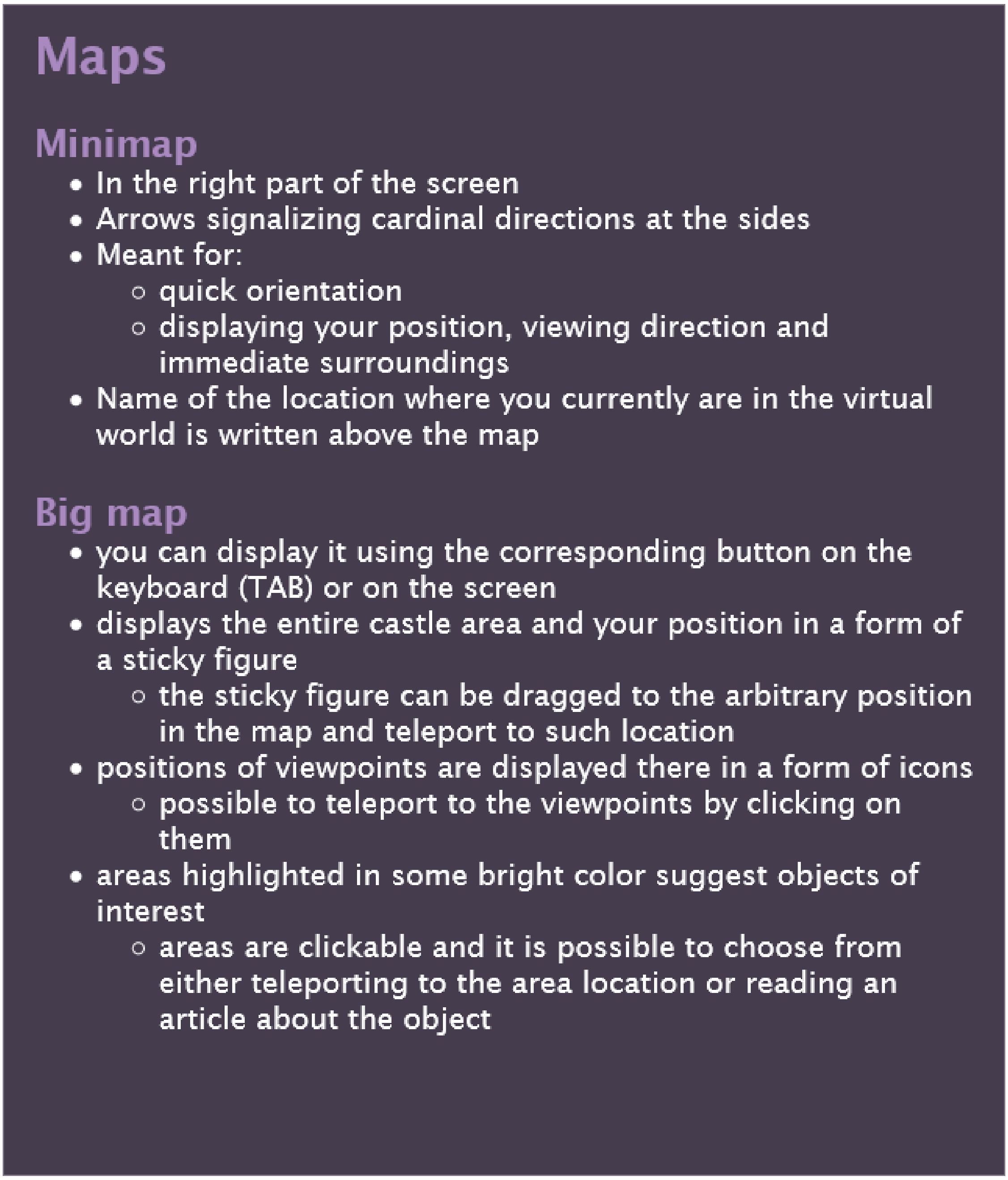
Walking and flying

- It it possible to walk on the ground or fly in the air
- Flying controls are a little harder to get than walking
  - it is often necessary to look around using the mouse and navigate the view to the place where we want to fly and perform a move forward
  - for changing flying altitude, use PAGE UP/DOWN keys or buttons on the screen

Viewpoints

- A place in the virtual world
- Interesting sight to the surroundings or an interesting object
- Possible to instantly teleport to its location

Figure E.13. User Guide 4/6 – Interaction, Walking/Flying, Viewpoints



Maps

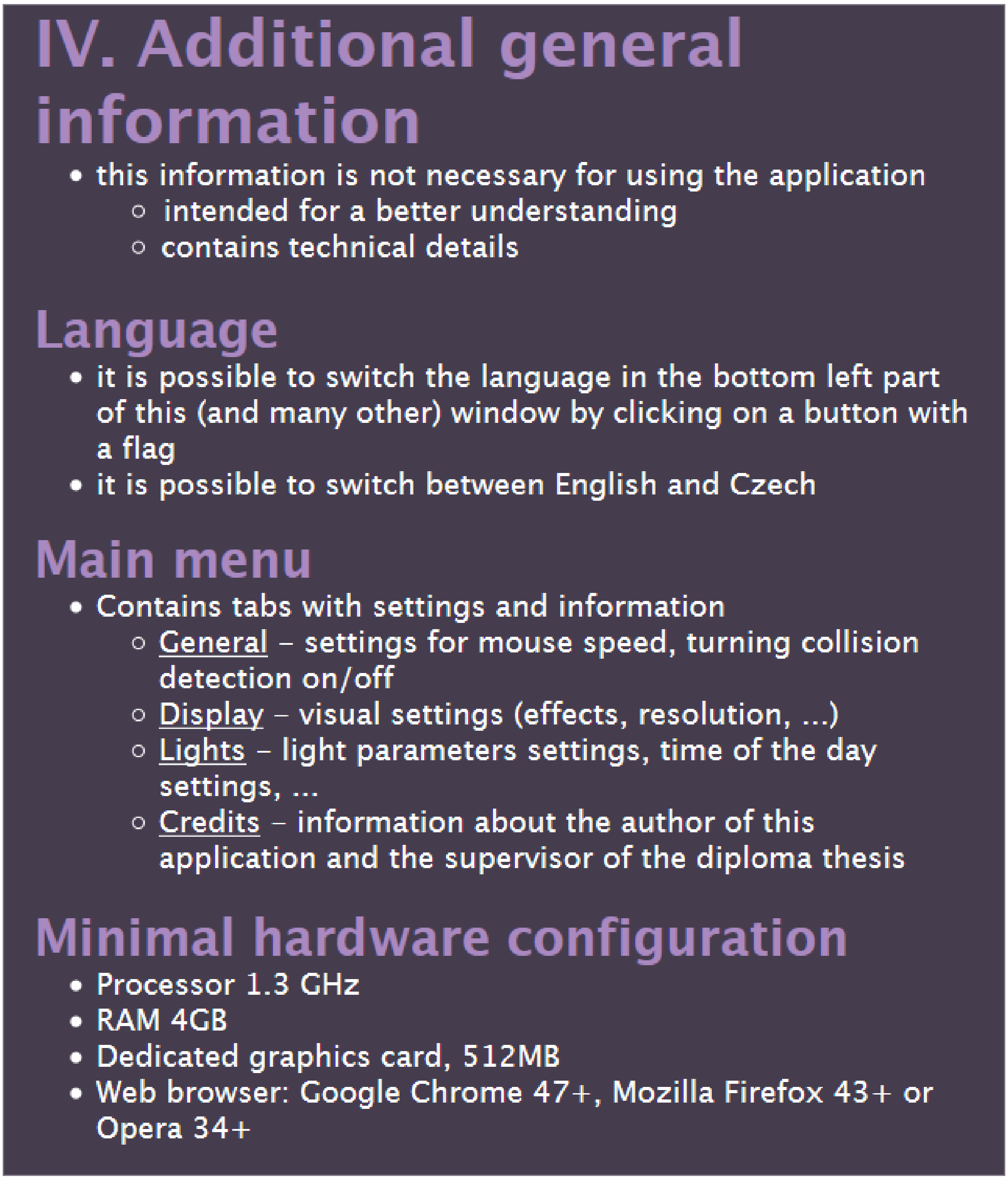
Minimap

- In the right part of the screen
- Arrows signalizing cardinal directions at the sides
- Meant for:
    - quick orientation
    - displaying your position, viewing direction and immediate surroundings
- Name of the location where you currently are in the virtual world is written above the map

Big map

- you can display it using the corresponding button on the keyboard (TAB) or on the screen
- displays the entire castle area and your position in a form of a sticky figure
    - the sticky figure can be dragged to the arbitrary position in the map and teleport to such location
- positions of viewpoints are displayed there in a form of icons
    - possible to teleport to the viewpoints by clicking on them
- areas highlighted in some bright color suggest objects of interest
    - areas are clickable and it is possible to choose from either teleporting to the area location or reading an article about the object

Figure E.14. User Guide 5/6 – Maps



IV. Additional general information

- this information is not necessary for using the application
  - intended for a better understanding
  - contains technical details

Language

- it is possible to switch the language in the bottom left part of this (and many other) window by clicking on a button with a flag
- it is possible to switch between English and Czech

Main menu

- Contains tabs with settings and information
  - General – settings for mouse speed, turning collision detection on/off
  - Display – visual settings (effects, resolution, ...)
  - Lights – light parameters settings, time of the day settings, ...
  - Credits – information about the author of this application and the supervisor of the diploma thesis

Minimal hardware configuration

- Processor 1.3 GHz
- RAM 4GB
- Dedicated graphics card, 512MB
- Web browser: Google Chrome 47+, Mozilla Firefox 43+ or Opera 34+

Figure E.15. User Guide 6/6 – Language, Main Menu, HW requirements

Appendix F
Contents of the enclosed DVD-ROM

The DVD contains following directories and files:

- `application/`
  - source codes of the application (HTML, CSS, JavaScript, GLSL)
  - application executable (`index.html`)
- `thesis/`
  - `DT_Smrcek_Antonin_2016.pdf`
    - text of the thesis (`PDF`)
- `documentation/`
  - `program/`
    - contains programming documentation for the source files
  - `vpc_content_completion_guide.pdf`
    - guide for developers which explains process of addition of a new virtual content to the application
- `content/`
  - `models/`
    - `main_scene/`
      - `prague_castle.skp`, `prague_castle.dae`
        - virtual scene with all models (SketchUp and Collada formats)
    - `terrain/`
      - `terrain.dae`
        - model of the environment around the castle
      - `terrain.png`
        - heightmap which was used for generating the terrain
  - `textures/`
    - contains all created textures (`JPEG`)
  - `photos/`
    - contains all taken high-resolution photographs (`JPEG`)