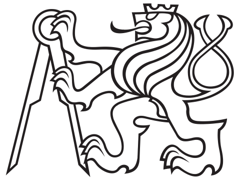
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Efficient Rendering of Earth Surface for Air Traffic Visualization

Vojtěch Kaiser

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
January 2018

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Vojtěch Kaiser

Studijní program: Otevřená informatika
Obor: Počítačová grafika a interakce

Název tématu: Efektivní zobrazování zemského povrchu pro vizualizaci letecké dopravy

Pokyny pro vypracování:

Proveďte rešerši metod pro škálovatelné zobrazování letecké dopravy v planetárním měřítku. Zaměřte se na metody pro zobrazování rozsáhlých terénů. Zvolte vhodnou metodu pro vykreslování Země s reálnými daty reprezentujícími zemský povrch. Zaměřte se na optimalizaci vizuální kvality výstupu za předpokladu omezeného množství dat uchovávaného v hlavní paměti. Umožněte zobrazování obecných 3D objektů (modely letadel a letišť) a jejich popisků. Implementaci zvolených metod realizujte jako nový vizualizační nástroj pro systém AgentFly. Výslednou implementaci důkladně otestujte v několika vizualizačních scénářích a s použitím různých hardwarových platforem.

Seznam odborné literatury:

[1] R. Kooima, J. Leigh, A. Johnson, D. Roberts, M. SubbaRao and T. A. DeFanti: Planetary-Scale Terrain Composition. In IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 5, pp. 719-733, Sept.-Oct. 2009.

[2] A. Mahdavi-Amiri, T. Alderson, F. Samavati: A Survey of Digital Earth, Computers & Graphics, Volume 53, Part B, December 2015, Pages 95-117, ISSN 0097-8493

[3] F. Losasso, H. Hoppe: Geometry clipmaps: terrain rendering using nested regular grids. ACM Transactions on Graphics (TOG). Vol. 23. No. 3. ACM, 2004.

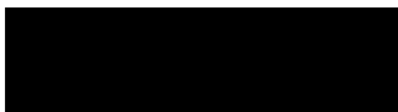
[4] Livny, Y., Kogan, Z. & El-Sana: Seamless patches for GPU-based terrain rendering. J. Vis Comput (2009) 25: 197.

[5] M. Wimmer, P. Wonka: Rendering Time Estimation for Real-Time Rendering. Proceedings of Eurographics Symposium on Rendering 2003, ACM SIGGRAPH, June 2003.

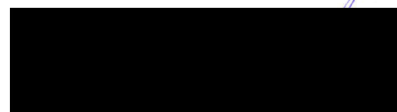
[6] O. Mattausch, J. Bittner, M. Wimmer: CHC++: Coherent Hierarchical Culling Revisited. Comput. Graph. Forum, 27(2), pp. 221-230, 2008.

Vedoucí: doc. Ing. Jiří Bittner, Ph.D.

Platnost zadání: do konce zimního semestru 2018/2019



prof. Ing. Jiří Žára, CSc.
vedoucí katedry



prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 3.4.2017

Acknowledgements

I would like to express my gratitude to my supervisor Jiří Bittner for his useful advice, comments and remarks. Furthermore, I would like to thank my parents for all the support, emotional and financial, they provided me with. At last, I would like to express my sincere gratitude to my dear Dominique for keeping me sane, encouraging me and showing me all the support possible, emotional and grammatical.

Declaration

I hereby declare that I have completed this thesis independently and that I have used only the sources (literature, software, etc.) listed in the enclosed bibliography.

Prague, 9. January 2018

Abstract

When rendering large-scale scenes, we do encounter a different set of issues than in scenes usually used in games or cinematography. These need to be addressed at the lower level of used visualization engine. In case of this work, the problem is to render Earth surface for any camera configuration, maximizing visual fidelity while avoiding framerate stutters. We break our large data-set into a hierarchical data structure and then use incremental scene construction algorithm that builds a slice through the data structure, taking into account cost and fidelity of particular elements. The final implementation maintains steady 60 frames per second while visualizing Earth surface from any distance or direction. The devised approach works for intended use cases in the AgentFly simulation system, although it needs more development in the future.

Keywords: rendering, large scenes, planet, geospatial visualization, LOD, Java, OpenGL

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Při vykreslování rozsáhlých scén se potýkáme s jinými problémy než v případě scén pro kinematografii nebo počítačové hry. Těmto problémům se musíme věnovat na nižší úrovních vizualizačních enginů. V případě této práce řešíme vykreslování povrchu země pro libovolnou konfiguraci kamery, a snažíme se maximalizovat vizuální kvalitu vykreslených snímků vyhýbající se viditelným poklesům ve vykreslovací frekvenci. Naše vstupní data jsme rozložili do hierarchické datové struktury, a s pomocí inkrementálního algoritmu stavíme řez touto hierarchií s ohledem na vizuální kvalitu a cenu vykreslení jednotlivých elementů. Představená implementace udržuje stabilních 60 snímků za vteřinu při vykreslování povrchu země z kterékoliv vzdálenosti a směru. Výsledný přístup funguje pro všechny uvedené scénáře použití pro AgentFly simulační systém, avšak další vývoj je nezbytný.

Klíčová slova: vykreslování, rozsáhlé scény, LOD, planeta, Java, OpenGL

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Structure of this thesis 2

2 Background **5**
2.1 The current system 5
2.2 General requirements 8
2.3 Functional requirements 8
2.4 Expected use cases 9
2.4.1 Large scale Earth 10
2.4.2 Small scale Earth 12
2.4.3 Controller view 13
2.5 Related work 14

3 Scalable Earth surface representation **17**
3.1 Quad tree 17
3.1.1 Indexing 18
3.2 Elevation 20
3.2.1 Format 20
3.2.2 Sources 22
3.2.3 Query 23
3.3 Imagery 25
3.3.1 Format 25
3.3.2 Sources 26
3.3.3 Sentinel 2 28
3.4 Mesh 31
3.4.1 Generation 32
3.4.2 Decimation 33
3.4.3 Stitches 35

4 Rendering Earth surface **39**
4.1 Spatial precision 39
4.1.1 Double precision matrices . . . 41
4.1.2 Dynamic center method 41
4.1.3 Fixed grid center 44
4.1.4 The selected option 45
4.2 Slice construction 46
4.2.1 Visual fidelity model 46
4.2.2 Algorithm base 47
4.2.3 Algorithm extension 47
4.2.4 Desirability heuristic 48
4.2.5 Adjacency enforcement 49
4.2.6 View frustum culling 50
4.2.7 Occlusion culling 51
4.3 Load balancing 52
4.3.1 Data transfer 53
4.3.2 Fidelity settings 55
4.3.3 Memory limits 56
4.4 Controls 57
4.4.1 Pan 57
4.4.2 Zoom 59
4.4.3 Look around 59
4.4.4 Inspection point 60

5 Implementation **63**
5.1 Technology 63
5.2 Engine organization 64
5.3 Render passes 66
5.4 Dynamic shaders 67
5.5 Render batching 68
5.6 Integration 70
5.7 Scenarios 70
5.7.1 Earth layer provider 71
5.7.2 ATC view 71

6 Results **77**
6.1 Missing features 77
6.2 Known issues 79
6.3 Generator performance 80

6.4 Earth viewing performance 82
6.4.1 Fixed quality 83
6.4.2 Comparative 85
7 Conclusion **89**
7.1 Future plans 90
Bibliography **93**
A CD contents **95**
A.1 Files 95
A.2 Distribution 97
A.2.1 Data 97
A.2.2 Execution 97
A.2.3 Controls 98
A.2.4 Outputs 99
B Measurement details **101**
B.1 Tile generator performance . . . 101
B.2 Earth viewing performance . . . 101
B.3 Tested computer 101
C Considered technology **105**
C.1 Engines 105
C.2 Additional technologies and data sources 108

Figures

2.1 Data transfer approaches 7
2.2 VFR and satellite maps 10
2.3 Sector visualization 11
2.4 Flight plan tunnel 11
2.5 Wind visualization 12
2.6 Prague airport 12
2.7 Scanned model 13
2.8 Controller view 14

3.1 Indexing example 18
3.2 Indexing relations 19
3.3 RGB elevation tiles 21
3.4 3" arc coverage 22
3.5 1" arc EU coverage 23
3.6 1" arc USA coverage 23
3.7 Elevation map normal 25
3.8 Blue Marble top level 26
3.9 Black Marble top level 27
3.10 Landsat 7 top level 28
3.11 Sentinel 2 top level 28
3.12 PlanetSAT top level 29
3.13 Sentinel 2 coverage 29
3.14 Sentinel 2 color correction 30
3.15 Sentinel 2 coastline cutoff 31
3.16 Sentinel 2 loud coverage 32
3.17 Elevation sample mapping 33
3.18 Average plane calculation 34
3.19 Earth mesh regular 35
3.20 Earth mesh decimated 35
3.21 Tile mesh stitch 36

4.1 Single precision transformation . 43
4.2 Offset transformation 43
4.3 Double precision transformation 44
4.4 Fixed grid 45
4.5 OBB frame, e = 100 51
4.6 OBB frame, e = 1 52
4.7 Occlusion cone construction 53
4.8 Occlusion cone test 54
4.9 Camera pan 58
4.10 Zoom calculation 60

5.1 Render batching 69
5.2 Earth, no exaggeration 72
5.3 Earth, no lights 73
5.4 Earth, e = 100, global 73
5.5 Earth, e = 100, d = 6 74
5.6 ATC view, global 74
5.7 ATC view, closeup 75

6.1 Generator performance 82
6.2 Frame time, Q = 1 84
6.3 Dissatisfied, memory, Q = 1 85
6.4 Cut size, rendered, Q = 1 85
6.5 Frame time, comparative 86
6.6 Dissatisfied, comparative 86
6.7 Cut size, comparative 87
6.8 Rendered, comparative 87

B.1 Frame time, Q = 0.025 102
B.2 Dissatisfied, memory, Q = 0.025 102
B.3 Cut size, rendered, Q = 0.025 . 103
B.4 Frame time, Q = 5 103
B.5 Dissatisfied, memory, Q = 5 . . 103
B.6 Cut size, rendered, Q = 5 104

Tables

3.1 Dataset comparison 29

6.1 Test PC details 83

B.1 Tile generator times 101

B.2 Additional test PC details 104

C.1 Engine comparison table 108

Chapter 1

Introduction

This thesis focuses on design and implementation of replacement of visualization system in the project AgentFly. The currently used system, *visio*, has become obsolete over the years, and its replacement is more than necessary. The following pages will cover the current state and reasons it needs to change, analysis of used data for visualization, rendering techniques for large-scale scenes such as those in air traffic control, and finally the implementation of proposed solutions.

Realization of the engine itself will be covered only lightly, as our primary interest is the visualization of Earth surface with visual fidelity being adjusted to fit the current point of view. The topic of visualization engines is very well covered, and we will focus on what needs to be different to satisfy specific needs of planetary scale scenes.

1.1 Motivation

AgentFly [1] is a system that operates in two spheres of interest. Civilian air traffic control simulation and analysis, and various scenarios around unmanned aircraft control. The former requires visualization of global data over Earth surface and ATC view screens, while the latter uses mostly small scenes with no significant issues.

The old visualization system turned obsolete for reasons discussed in the following text, from which the most important is spatial precision around the surface, and overall rendering performance. In essence, visualized scenes get rendered under ten frames per second and vertices shake due to floating point precision issues when the camera is close to the surface of the Earth.

Because the whole system is built on technology that is no longer supported is beyond saving, new one needs to be implemented to replace it. This means a new visualization engine that has the same functional capabilities as the old one needs to be constructed (or adapted) while solving our precision and

[1] `www.agentfly.com`

performance issues.

Since the main problems in the previous system were with large-scale scenes with the entire Earth, that will be our focus when implementing the replacement. We need to deal with adaptive visual fidelity from a large dataset of textures (satellite imagery) and geometry (meshes from elevation maps) while maintaining sufficient spatial precision (non-trivial, as graphics pipeline operates in single precision floats).

When changing such an integral part of our work pipeline, we need to take a great care designing the new architecture. This goes hand in hand with integration of the new system in the rest of the AgentFly project, as many modules are intertwined with the old solution, and amount of work to transfer will be non-negligible.

1.2 Structure of this thesis

This thesis is composed out of four main parts that will gradually take us from the current state being replaced to working implementation of the replacement.

In the "background" chapter current visio system will be discussed and what exactly is right or wrong with it. Then we will focus on formal requirement specifications that lay out boundaries of our design, mostly based on shortcomings of the current solution. To help us grasp the scope and the use of the new system, we will take a look at most prominent use cases of previous one, whether they are being used or are planned to be implemented. At last, we will explore the related work on large-scale scenes and planetary scenes with real data.

Before investigating ways to visualize Earth surface, first we need to define what data and format will be forming it. The reasoning behind quadtree choice will be unveiled, followed by its application on elevation data and satellite imagery. The elevation map data structure is going to be defined as an API with a description of its queries and their uses in surface shape generation, while satellite imagery datasets will be discussed from accessibility and quality standpoint. In the end, scene object generation is going to be formalized in the form of quadtree tile structure and relation between these tiles.

When rendering large-scale scenes, we need to address spatial precision that stems from floating point arithmetic issues. Then we will move onto the construction algorithm, discussing various optimizations that will help us build the best scene for a specific view. Further options for load balancing will be elaborated. With the scene built and optimized for maximum performance, ideal camera controls for common use cases are going to be proposed.

In the last part the implementation of the new system will be reported. Specific optimizations will be described and placed in the context of the

whole system. Two example scenarios will be introduced, one for air traffic controller view, and the more crucial one, for global Earth view. Before moving to performance reports and benchmarks, known issues and missing features will be listed to contextualize these results.

Chapter 2

Background

The following sections will cover the current system, its usage, and requirements we can conclude from it. We will take a look at current and expected use cases to highlight what features are the most important considering a complete transition from the old system to its replacement.

2.1 The current system

Firstly, we need to take a closer look at details of the current system. Its inner design should hint us what mistakes can be avoided, and its current use cases should be a hint where we can gain more performance in comparison to general or game visualization systems.

The system we will be looking at, further referenced as *visio*, was developed under *java3d* [1] visualization API, which was at the time one of the few well supported robust options with ongoing development.

One of the key issues with visio is that java3d, despite being developed by Oracle, stopped being officially supported around 2008, and unofficial support from various forks definitely ceased around 2012. Since java3d suffers not only performance-wise but also lacking in features, it makes it less and less practical each passing year.

Lack of support is not only reason java3d is no longer suitable for AgentFly, as Oracle replaced it with JavaFX scene graph. The whole idea of the standard game engine or scene graph does not fit massive asynchronous visualization, and position precision issues would be challenging to resolve practically in any engine.

Over the years, visio performance did not rise, and at the moment, it runs somewhere around ten frames per second. Various bugs and issues started popping up, for instance, the maximum line width of two or broken painter's algorithm for transparent objects (change of sorting algorithm from

[1] `https://en.wikipedia.org/wiki/Java_3D`

Merge-Sort to Tim-Sort in Java 7).

Taking a look at implemented features of current visio, we do not see much out of the ordinary. Of course, advanced materials and textures are missing, as is shadow rendering, but these were never really necessary for any scenario in the development.

Approach to data. As far as engine built over *java3d* goes, the solution is actually quite fitting the use even today, but there was a line of intended usage changes, and in the end, none of them work particularly well for their own reasons.

First introduced was so called LayerProvider construct, where a wrapper class took data from simulation and processed it into its own scene graph. Each simulation scenario then defined a list of these providers to be used for its visualization window. This approach was the most potent, as pretty much all features of the system could be accessed in it.

Second, possibly because of the introduction of distributed simulation, came the idea that visualization could run on a different computer from the simulation. This was implemented through the system of VisualModule classes, where all these modules have common implementation LayerProvider counterpart. The scene graph is then created right where visualized data are created, and then it is sent in the form of commands to its counterpart (possibly over the network). The feature set of visual modules is somewhat simplified, for instance, user input access was reduced to on/off toggle controlling the visibility of the whole thing.

The last iteration came in the form of DynamicVisualModule. Since most of the layers were not visible during usage, network and event channels were flooded with unnecessary graphics updates. Dynamic visual modules added the subscribe/provide system that sends data only when someone is willing to draw them. This module was meant as a direct replacement of VisualModules but got even less attention with implementation, and only the most talkative layers were changed into it to solve the immediate problem, and with that, only features these used were implemented in it.

The sensible solution thus seems to be sending data to visualization, as it was in the first implementation of layer providers, but only when there are any subscribing to it. This would combine the best of both worlds. With good data fractioning, the same information can be used by many different providers, which is not possible with visual modules and data have to be sent multiple times.

Figure 2.1 shows above approaches to data transfers in the order they were described. Note the message texts, primarily for the difference between data and scene being sent. The layer provider entity represents an interface covering distinct implementations for each case.

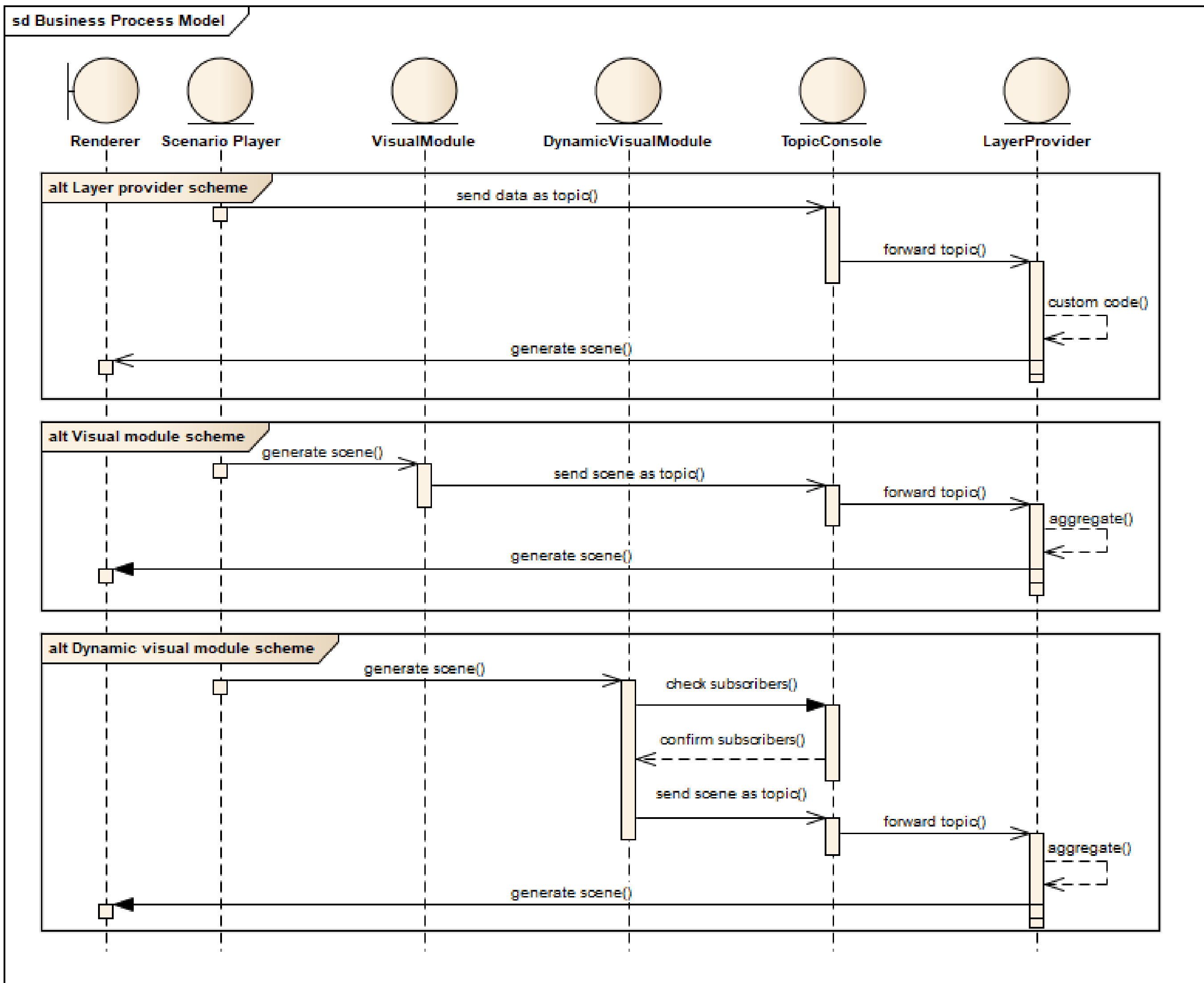


Figure 2.1: Sequence diagram for data flow in three patterns in current system.

Optimizations. The only deliberate optimizations of rendering speed in visio are done in (dynamic) visual modules, where received objects are sorted into prepared aggregation objects to reduce the number of draw calls. It is important to note that objects coming into this process (user scene graph) are the same as the ones coming out of it, so no platform dependent knowledge is being exploited to speed up the rendering process.

This approach could be a good thing if we were to switch the underlying technology quite often and thus any optimizations would carry over, but since we have a single technology of choice for years, it makes sense to move them under the hood.

2.2 General requirements

General requirements, or sometimes referred as non-functional, are such, that place constrains on how the system should be implemented, rather than what specifically. These rarely change and should represent a broad idea of how will the final product behave.

Multi-platform New implementation should run on most recent versions of Windows, Linux, and OSx distributions.

Spatial precision The current visio has spatial precision in tens of meters, which is not enough for scenarios around airports and scenarios for UAVs. The new system should be significantly more precise; specifically, the margin of error should be below one centimeter.

Speed Even though current simulation is not utilizing as many threads as it could, and thus not utilizing CPU resources to the fullest, resulting code should use as little of CPU resources as possible, leaving it for simulation itself. The system should run well on lower-end GPUs, and sufficiently on integrated ones.

Extensibility Easy extension of visualization using a module-like system, abstracting interface of the system allowing silent technology modification, and have graphics providers written to fit data being displayed.

Synchronization The current implementation works in two modes, either graphics is created at a remote source and is sent to the device displaying it, or it is created from a local data source at displaying device. Synchronization with simulation in the new system should be unified and optimized.

Documentation The whole system should be well documented and described for reference in future development. The documentation should also include manual for users of the system on best practices for its use.

Scalability System should be prepared to handle well increasing amounts of displayed data.

Decoupling System should be able to handle multiple windows simultaneously, and also multiple computers participating in simulation or visualization.

Data sources Data should be available from offline and online sources.

Late-comer Visualization client connecting into ongoing simulation should be able to produce a valid scene.

2.3 Functional requirements

Functional requirements specify things that need to be implemented in the new system. These should broadly cover all topics of implementation, may change over time, but the overall scope of the project should stay the same. The following list represents a basic version of the new system, that should

serve as a foundation for upcoming development. Functional requirements that are implied from general requirements are not listed.

Complete scene graph The system should offer all essential tools for scene construction, such as layers, groups, transformations, or links.

Renderable objects Basic renderable objects should be implemented, such as lines, points, and meshes.

Background optimizations The system should analyze data to be displayed and look for ways to aggregate render calls without explicit user intervention.

Material-lighting Suitable renderable objects should have material and lighting attributes.

Data sources OBJ and MTL file format support.

Dynamically constructed Earth model Displayed Earth model should dynamically change its shape and fidelity based on camera position.

Height map integration The Earth model should integrate height maps.

Text rendering The system should provide efficient text rendering tools for text in world space and screen space.

Transparent meshes The system should incorporate blending for purely transparent meshes.

User input mapping User inputs should be mapped to appropriate actions and provided to user classes.

Screen capture The system will allow different modes of screen capture, such as video and images based on real-time manipulation with the scene, as well as based on simulation time based one.

Graphics primitives The system will support set of graphics primitives that can be placed in the environment.

Virtual Reality The system has the possibility to integrate VR for UAV operators, meaning two different cameras looking at the same scene in high refresh rates.

GUI integration The system should be designed with GUI framework in mind, as the final version of visio system should be encompassing complete development environment.

Asynchronous modification The scene should be modifiable independently on rendering cycle of the engine.

Since this work is not focusing on completion of the proposed system, it concentrates on laying out the foundation for its future development. First and foremost, the dynamic Earth will be our sole goal, and most other implemented features will be its prerequisites.

2.4 Expected use cases

In this section will be listed expected use cases of the new system, mostly overlapping with current use of visio with an addition of future projects.

2.4.1 Large scale Earth

It is necessary to render large-scale *air traffic control* (ATC) on actual 3D Earth due to distortion that occurs in any mapping to 2D space. For small-scale sectors is used stereographic projection, which configured to minimize the relative error, but that is simply not possible when looking at flights across the globe. This view consists of several major sections.

Earth surface. The first and foremost is the surface of the Earth itself, with fidelity changing based on distance from the camera. This surface can be either textured with satellite imagery, air traffic control maps, or tiles generated from a vector source such as open street maps. Two used textures are visible in figure 2.2. For the most part, this surface provides visual cues for quick orientation around the airspace, but it can contain any number of additional information. Even though the Earth itself is relatively smooth, we want to display elevation to have our traffic land/takeoff at the correct altitude. This elevation can be exaggerated to provide further visual cues.

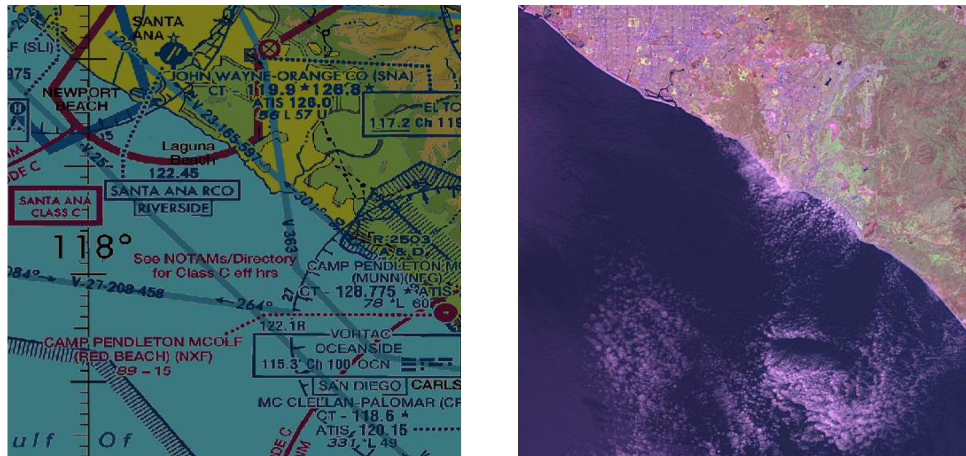


Figure 2.2: VFR map (left) and satellite image (right) used as surface textures.

Air control sectors. Representation of physical boundaries between controlled sectors has been so far visualized as colored semi-transparent volumes (figure 2.3). The number of these can be quite substantial, and it is not usual to have all centers and their sectors displayed at the same time unless there is some additional information. For instance, the color of sectors was previously used to visualize cognitive load of their controllers (figure 2.3).

Aircraft. The most important part — for simulation purposes at least — is, of course, the traffic itself. Besides displaying a model of the corresponding type (it is not realistic to have a model for each existing aircraft, so our primary goal is to map each aircraft to model with similar properties, such as size, number of engines, wingspan), we also want to display various data about that particular flight.

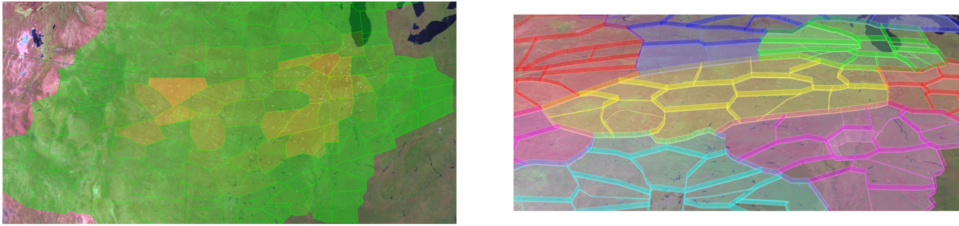


Figure 2.3: Sector boundaries used to display cognitive load of their controller (left), sectors under different centers (right).

Aircraft label (*datablock*) can contain the usual ground/horizontal/vertical speed, aircraft type, call sign, but also runtime properties such as currently running interaction of the pilot with ATC. The key point here is that every aircraft will carry around its data block with content that may completely change each simulation tick, and it should not *lag* behind (update of label position is one tick behind aircraft position update).

Now that we have an aircraft in space, we would like to know its direction and origin of departure. There are two kinds of flight plans being displayed, original schedule and up to date plan according to adjustments from ATC. The latter is visualized as semi-transparent tunnel as a five miles radius one flight level cylinder around each position on the plan (figure 2.4). This plan does not change all that often, but it can be quite long which can pose some issues from the rendering standpoint.

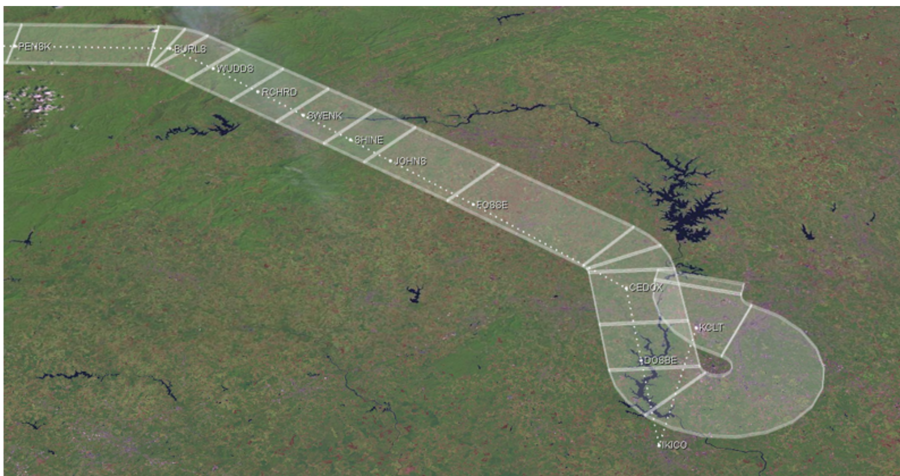


Figure 2.4: Aircraft flight plan visualization near an airport.

Weather information. ATCs also make their decisions based on weather conditions (mostly around airports), and we need to visualize such in the least intrusive way. For instance, ATC may choose a route that is more fuel efficient based on direction and strength of wind, so we currently display that as vector array, but it is entirely possible to aggregate more weather information in the on-the-fly generated texture. Figure 2.5 shows the current

visualization of wind data over the USA.

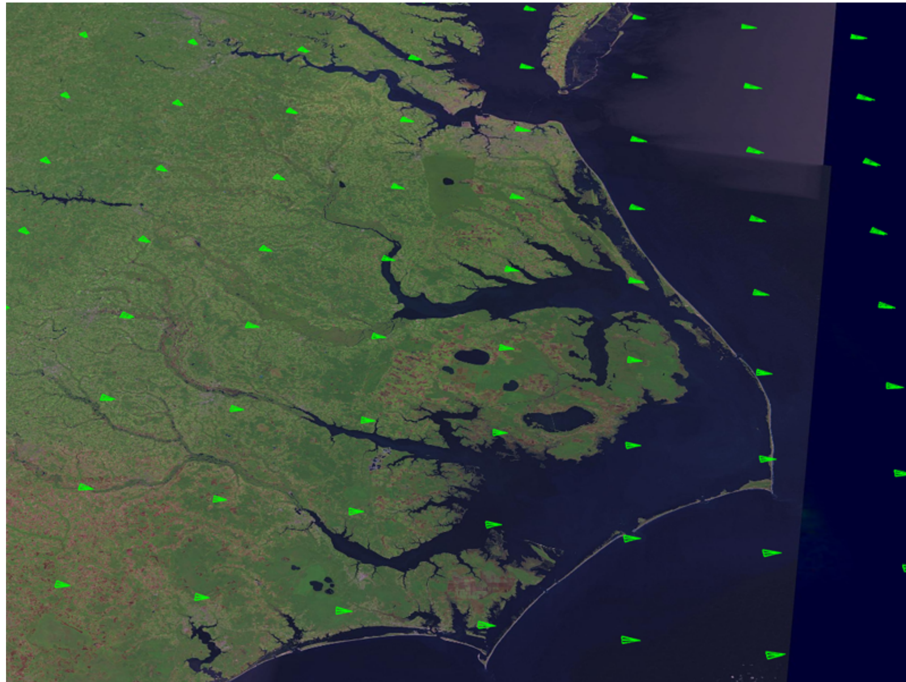


Figure 2.5: The current visualization wind data in the form of a grid of color coded arrows.

2.4.2 Small scale Earth

As small scale scenarios can be considered scenes of interest (hot spots) with a radius of at most several kilometers, such as airport traffic control, or basically any UAV scenario. Spatial precision is absolutely critical in these. Airport traffic control obeys the same set of requirements, with a small addition of airport buildings and detailed visualization of runways and taxiways (figure 2.6).

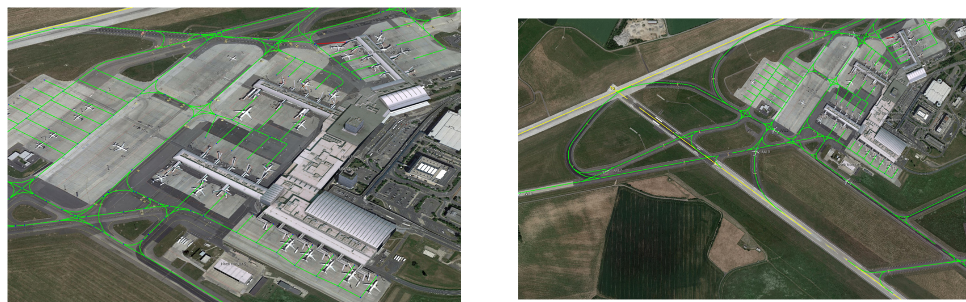


Figure 2.6: The current visualization of Prague airport with taxiways and buildings.

Buildings. Many UAV scenarios revolve around infrastructure, such as building 3D scanning or patrolling. When considering the latter, we can expect the simplest representations deprived of any photorealistic details, mainly because we are interested the most in colliders. Reconstruction from a video scan can, on the other hand, produce detailed geometry that needs to be handled equally well (figure 2.7).

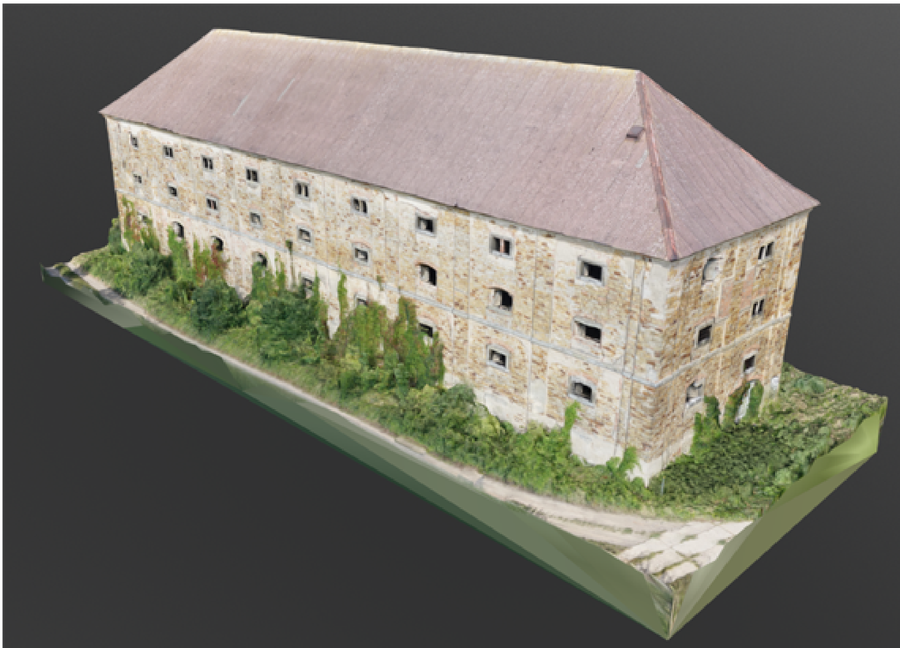


Figure 2.7: Model of a building scanned with UAV flyover.

Vegetation. When speaking of colliders, we cannot forget to mention trees. These can be placed manually in small areas, and there they behave like any other object, but can also be automatically generated in a larger scene. To increase the visual quality of rendered scenes, it is expected that grass coverage will be added at some point.

2.4.3 Controller view

The ATC view is the simplest use case of them all, as it uses only lines and text in orthographic projection. It consists of controlled sector outline, trace line with data block label for each aircraft, named fixes in that particular airspace, and a couple of inner screens with details on selected flights mostly in textual and table form (figure2.8).

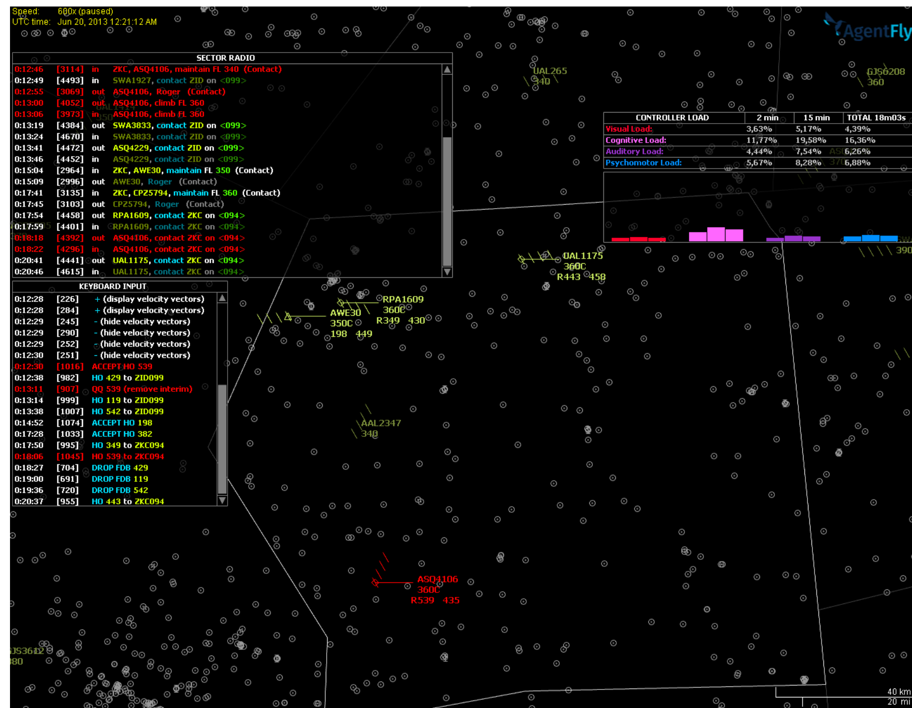


Figure 2.8: Example of controller view interface.

2.5 Related work

The visualization of large geospatial scenes comes with several problems to solve. Considering we already have data to display, likely in large quantities, we need to project them on the surface of the Earth. Snyder [3] aims to describe all possible map projections with all their details. These projections are categorized and with complete description of their features, usages, and necessary math for their application.

Having the data projected in a continuous space, we need to discretize them for rendering purposes and place them in a so-called Discrete Global Grid System (DGGS). Sahr et al. [1] is investigating most promising geodesic DGGS, but inspected systems are for the most part systems based on the icosahedron. Considering a DGGS of a particular structure, we need a way to index a cell within that is unique to its corresponding cell and can be used in both directions — storing and reading of location within the system. Mahdavi-Amiri et al. [4] is elaborating on all kinds of indexations in different DGGS. All above topics are also covered by another, more extensive, survey by Mahdavi-Amiri et al. [5].

Techniques used in adaptive triangulations can be split into three categories: regular hierarchical structures, general triangulations, and combined solutions. Regular ones can be represented by direct icosahedron subdivision as presented by Kooima et al. [15], square tiles that are subdivided into triangle patches

for texturing convenience discussed by Livny et al. [16], on the fly constructed geometric clip-maps introduced by Losasso et al. [17], and many more.

In Planet-Sized Batched Dynamic Adaptive Meshes by Cignoni et al. [2] (P-BDAM) is discussed interactive rendering of planet-sized textured terrain surfaces. It covers performance side of the problem, as well as floating point precision limitations (also covered by Thorne [10]). The used structure is regular triangle patch grid where each patch is irregularly triangulated to add elevation data. This work is further extended by Gobbetti et al. [13] (C-BDAM), adding compression and maximum visual error to the mix. It is important to mention the use of speculative prefetch as opposed to frame time prediction heuristics proposed by Wimmer and Wonka [14].

At last, we can represent and render the data as a triangulated irregular network (TIN). Instead of regular sampling, we place vertices in defining points on the surface, triangulating resulting point cloud, preferably as Delaunay (Kidner et al. [19]) or Quasi-Delaunay (Liu et al. [18]).

Chapter 3

Scalable Earth surface representation

Before we can start with the construction of any scenes Earth surface scenes, we first need to prepare the data in the best format for this use. In the beginning, we partition Earth surface in a hierarchical structure to subdivide the visualization problem, then we place displayed data in this structure, in our case elevations and satellite imagery, and lastly, we generate 3D objects that will be placed in a scene and rendered.

3.1 Quad tree

When looking for a hierarchical structure for Earth data, we need to simplify the problem a bit. Looking at the Earth as a spheroid makes things too complicated, especially if we do care only about its surface. The ideal solution is unwrapping it in 2D space, and continuing there, but no mapping from sphere to 2D is perfect, and all of them bring their own errors.

We could go with the stranger approach mapping on faces of a polyhedron [1] with further triangular subdivision, as used by Dutton [7] (octahedron) or Lee [8] (icosahedron), but mapping square textures into such a thing would be a nightmare. This brings up the first restriction, our mapping of choice should be *rectangular* to ease up the texture work as much as possible.

Looking for more common, conventional projection, we will find that Mercator projection [2] is rectangular, and as simple as it gets. We will merely treat latitude and longitude coordinates in degrees as values on y and x axis respectively, producing a rectangle with $2 : 1$ side ratio.

Now we just need to subdivide our Mercator rectangle to form a hierarchical structure. From structures with rectangular cells can be picked, for instance, a kD tree. This approach would efficiently cut out unnecessary detail over sea and poles, but it would make it difficult to get neighbors for cells we want to render (stitching purposes). To fix that, we can split the cell always in the

[1] http://www.progonos.com/furuti/MapProj/Normal/ProjPoly/projPoly.html
[2] https://en.wikipedia.org/wiki/Mercator_projection

middle, and alternate split axis regularly. We not only fixed neighbor lookup problem, but we also remove the main advantage of kD tree (adaptation to data), leaving us with a regular quadtree.

Quadtree in our case is slightly modified, as we have two zero level cells, western and eastern hemisphere. Otherwise, we split each cell into four smaller same size cells for the following level. Since root cells are square shaped (dividing 2 : 1 ratio rectangle), all cells in the tree are also square shaped.

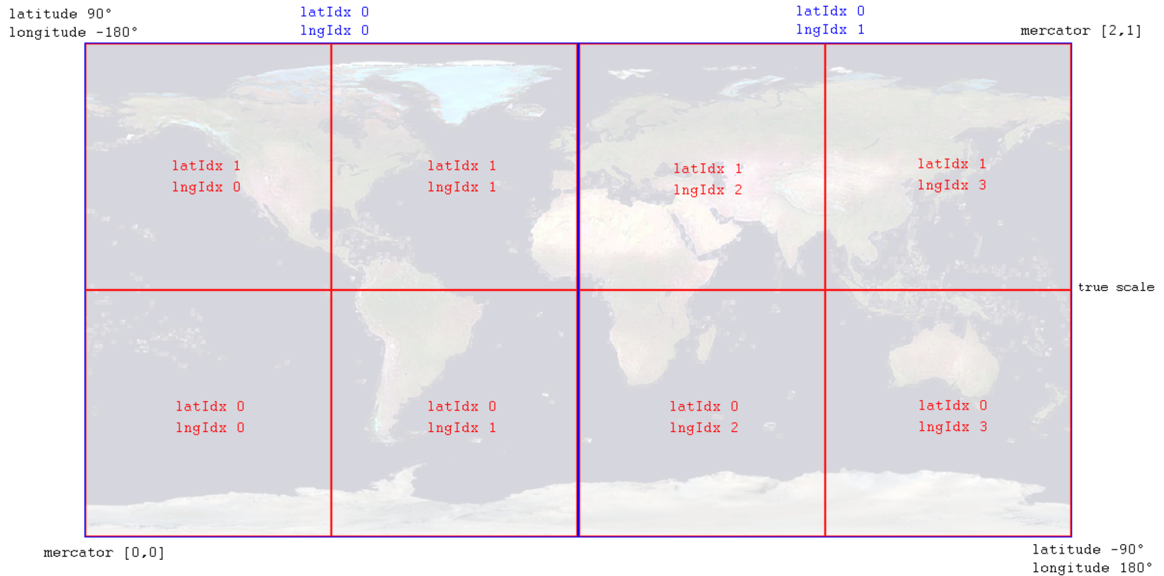


Figure 3.1: Indexed space at depth zero (blue) and one (red), mercator and spheric space bounds.

3.1.1 Indexing

To be able to deterministically name each node without having to build any structure, we need to devise an indexing method that allows us to perform all the necessary operations around a node. Methods to assign unique identifier to a cell are numerous, but they can be generally divided into three categories, as described by Mahdavi-Amiri [9].

- Hierarchy-Based indexing relies on numbering assigned to initial nodes in the structure, and the method of expansion of this code along subdivision. For example, we can name our root nodes A and B, and assign a number $0-3$ to each child. This way each cell is identified by a string following expression $[AB][0..3]^d$ where d is depth in the tree.
- Space-filling curve indexing, where we define a pattern of a walk through a cell grid, incrementing an address number. There is quite a few of possible curves (Hilbert, Peano, Morton, ...), each with different indexing complexity and use related other properties.

- Axes based indexing splitting addressed space by a number of axes and using a set of coordinates instead of a single number.

For our purpose, the best fitting approach seems to be axes based indexing. Index of a node in our quadtree is composed out of three numbers: latitude index, longitude index, and depth. To avoid the necessity to specify in which hemisphere are we indexing, the depth for latitude index will be increased by one in all calculations (figure 3.1).

We can imagine a regular grid over Earth surface, that has granularity dependent on depth. That means that grid at depth d will be a space $[0, 2^{(d+1)} - 1] \times [0, 2^d - 1]$, in which we index specific cell of interest. It is important to remember, that this space wraps between left and right edge.

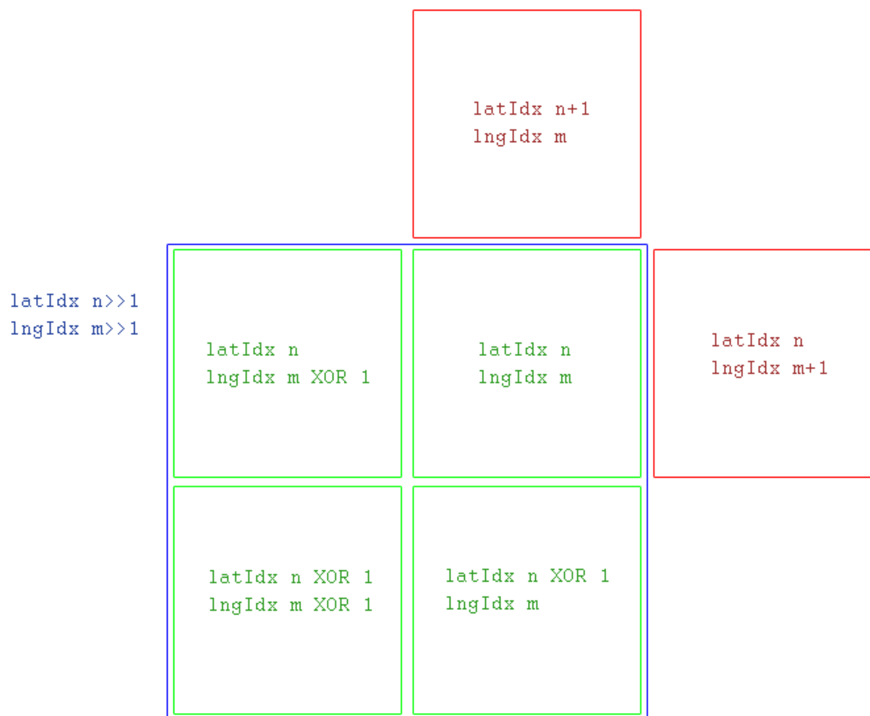


Figure 3.2: Relations between tiles, green siblings, red neighbors, and blue parent.

To demonstrate how simple is movement in between node indexes in the quadtree, basic operations will be briefly described below. See figure 3.2 for a visualization of some of the relations.

Ancestor. If we represent latitude or longitude index in binary, each bit corresponds to diving in to the left/right or bottom/top, respectively. Therefore, to get any ancestor in specific depth, we simply need to erase number of least significant bits. Consider being at depth d_1 with latitude index lt_1 going to ancestor at depth d_2 wanting to know its latitude index lt_2, where $d_1 > d_2$,

then $lt_2 = \left\lfloor \frac{lt_1}{2^{(d_1-d_2)}} \right\rfloor$. This may not seem overly simple, but in practice, going to parent means bit shift to the right.

Children. If we want to break down a node into its children, we can simply multiply its latitude/longitude indices by two, and add one if we want a child in the right column or top row.

Siblings. Distinguishing the difference between sibling and a neighbor is important. Sibling of a node has the same parent and always exists, while neighbor may have distinct parent and may not exists. It is important to note, that siblings may be missing in general quad tree. However, since we do not allow parent and children to coexist in single scene, and we demand full coverage of the surface, our quadtree will require presence of either none or all children. To get a sibling, we only modify the least significant bit, so we get the index of a sibling in a row as $lngIdx \oplus 1$ ($\oplus$ is bitwise XOR operation).

Neighbors. When considering siblings, we need to add extra checks for top and bottom to account for limited space on y axis and wrap on x axis. Since getting neighbors is movement in same grid granularity, we can simply add or subtract one, and apply space limitation checks. A neighbor does not have to exist, for instance, no node with latitude index equal to zero has the bottom neighbor, but it may be missing for other nodes due to undefined source data as well. We will consider nodes to be neighbor only if they share an edge.

3.2 Elevation

We will take a look at elevation data for the Earth, first in their raw form, then in the context of our quadtree structure, and finally their use case in our system.

3.2.1 Format

When discussing about elevation data, we usually consider some collection of samples with values in meters above/below sea level, organized depending on their usage. The organization can be dependent on the way the data were collected, for instance, the stripe tracing path of a satellite where rows are particular scan lines [3], or it can be dependent on the usage, such as rectangle area of samples corresponding latlong rectangle in Mercator.

Either way, we need to refit the data into our quadtree for further processing. That means writing a reader for each data source, which will find the best depth in the quadtree to scale the data to while minimizing sample and precision loss and convert all source samples to it. At this point, we have

[3] see `https://earthexplorer.usgs.gov/` for SRTM dataset samples

one consistent new set of tiles (quad tree nodes), which we want to reflect upwards towards the root (assuming previously present data were built from coarser samples). At last, we need to fill in missing data in all new tiles to avoid unnecessary edge cases during usage.

Going through this process will leave us with a somewhat large dataset of 2D float arrays that we need to store, but that would take way too much space and slow down downloading during usage. We could, of course, apply any standard compression such as *zip* or *rar*, but we can do much better.

The more compression algorithm knows about compressed data, the better it can compress it. That leads us to compression of 2D data, which is commonly applied to images. We simply map our float samples to colors in an image and store it using one of the conventional formats. Mapping to greyscale is not sufficient even if we resort to 16-bit tiff, so we will have to map samples to RGB space (figure 3.3). Consider sample s in meters which we want to convert to color c as single integer value containing 24-bit RGB.

$$c = round((s + 1000) \cdot 100) \cdot 16$$

We first offset original value by 1000 to get the range of values above zero, then we extract centimeters by multiplying by 100 and rounding, and at last, we shift it bitwise by four to the left to get least significant bits to the visible part of blue for visual inspection. This scheme leaves us enough space to record any elevation on Earth with centimeter precision (minimum -1000m, maximum 9000m).

Image generated with these colors can be stored using any lossless image file format (jpg turned out generating an average error of 600 meters).

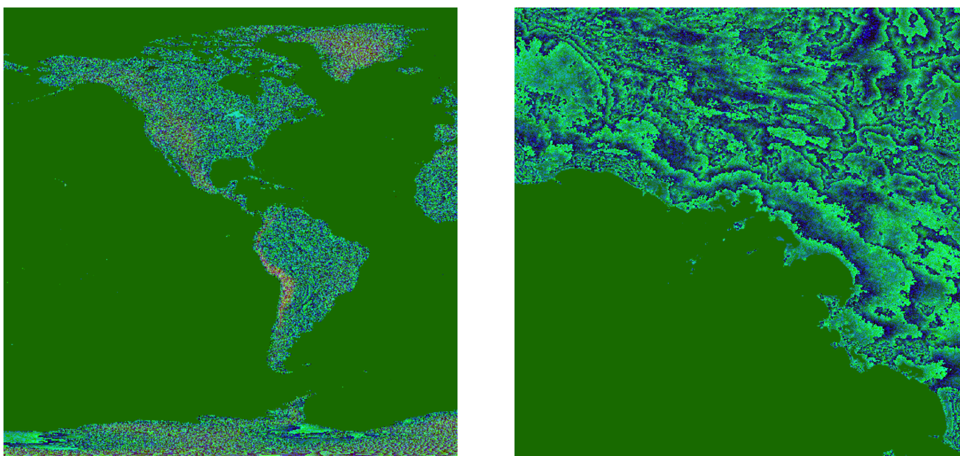


Figure 3.3: Example of zero depth (left) and depth six (right) elevation map tiles

3.2.2 Sources

As datasets for this project had to be used only ones under a license allowing commercial use. Two main sources were used, but one was partially source for the other. The resolution of sources is quantified in arc seconds per sample. That means what resolution was the dataset captured before its transition in Mercator projection, so even though there are more samples per meter in Mercator further from the equator, it is no more precise. The three levels defined are 15 arc seconds (460m/sample), 3 arc seconds (90m/sample), and 1 arc second (30m/sample). When talking about data constructed from SRTM, we should keep in mind that all samples were captured at 1 arc second and then transformed into lower resolutions based on the amount of missing data and fragmentation.

ViewFinderPanoramas. Well known and used collection of elevation data [4] in all three described resolutions. The author claims it is a collection of multiple sources with cleaned up missing samples and fragments. While 3 arc second dataset is complete mosaic for all land masses (figure 3.4), 1 arc second is available only for a couple of mountain ranges in central Europe and northern Europe (figure 3.5).

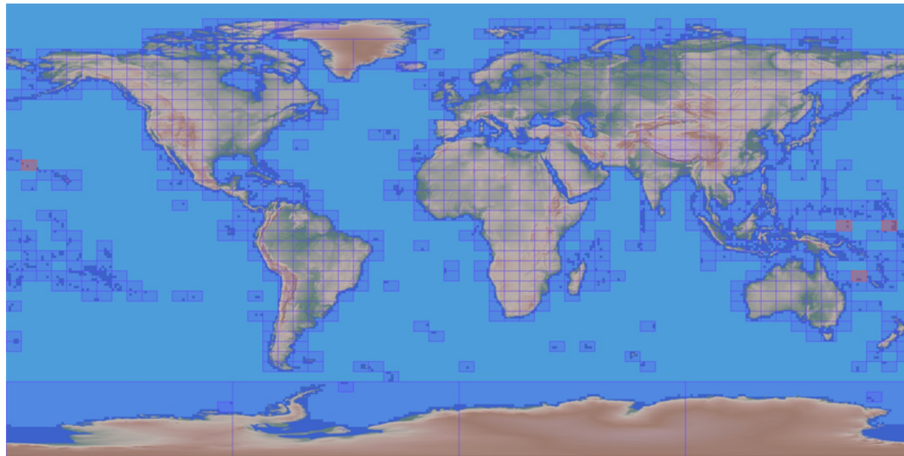


Figure 3.4: Map of 3 second arc global coverage.

SRTM. Shuttle Radar Topography Mission was a collaborative project under NASA aiming at global landmass elevation map coverage (more on the project in its documentation [5]). Specifically 1 arc second available for download from NASA servers [6] has coverage only over USA territories, and it is unfiltered mosaic, meaning there are missing data and fragments (figure 3.6).

[4] http://viewfinderpanoramas.org/dem3.html
[5] https://dds.cr.usgs.gov/srtm/version2_1/Documentation/SRTM_Topo.pdf
[6] https://dds.cr.usgs.gov/srtm/version2_1/SRTM1/

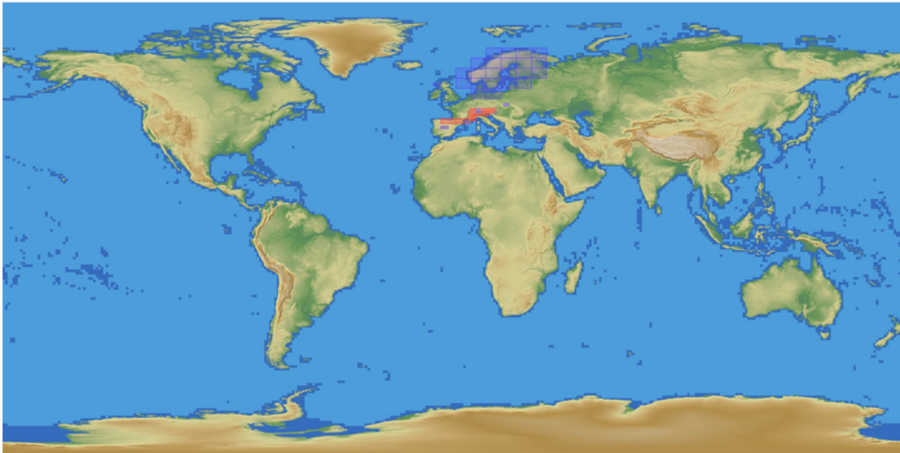


Figure 3.5: Map of 1 second arc EU coverage.

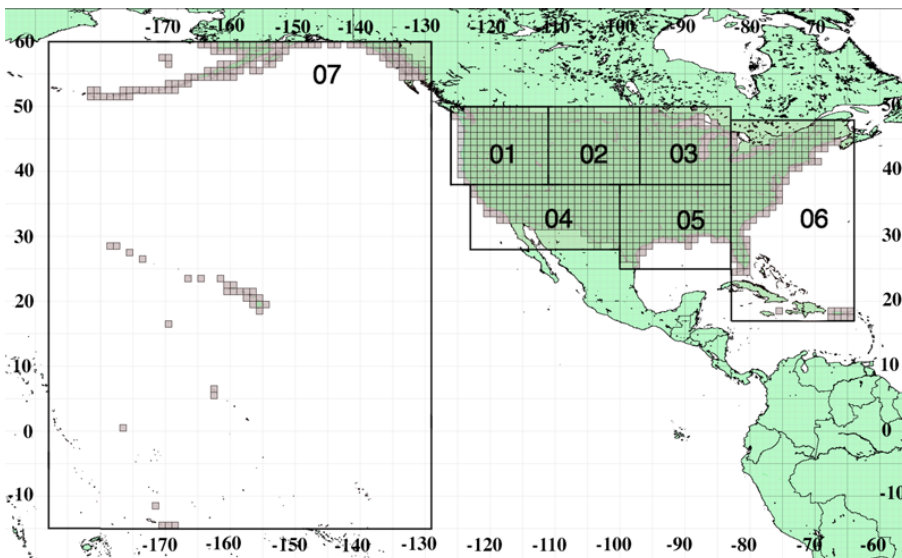


Figure 3.6: Map of 1 second arc USA coverage.

Connected. All datasets put together form a quadtree starting at depth 9 with full coverage going to depth 11 with selective coverage (USA, Alps, Northern Europe). The final size is 170GB over 480 000 tiles in 512 × 512 samples, zero tiles removed. All missing data were pulled from 15 arc second dataset, as it had complete coverage.

3.2.3 Query

Performing an elevation query is trivial when one does know exact tile in question, and as long as the query point is far from the edge of that tile. It gets a tad bit more complicated once we start querying elevation data based on latitude and longitude in degrees with no knowledge of how deep is the quadtree at that point.

Nearest Neighbor. Doing an NN query is relatively easy, as we just recursively dive into the quadtree from the following top position of query point to depth that interests us. This approach has one crucial flaw: if the query point is on the edge of a sample or even a tile, result of such search depends purely on float precision "randomness" how we got the query point. We can calculate the same point twice using different methods (calculating an edge point on two different levels involves different starting numbers, for instance), and the resulting elevation may differ significantly.

Linear interpolation. To make sure our query process is stable, we need to do four NN queries and linearly interpolate between them. That way, a slight difference in query point will result in a slight difference in returned elevation, and that can be usually tolerated. To make our four queries, we need to go to the grid at our depth of interest, take closest cells around query point, and use their centers for NN search.

We can, of course, perform a range search around initial query point, but that gets a little complicated around edges and corners of tiles, as relevant samples can be far depth-wise. Range search approach would potentially save us significant number of traversal steps.

Sampling depth. Mentioned several times, we have to deal with "depth of interest". This depth can depend on use; for instance, we want all queries to be performed at the same non-maximum depth when constructing tile stitch at that depth to avoid aliasing. We can also want to get elevation from the maximum depth not knowing how much it really is. The former is easy to resolve, as we perform NN queries in the grid of depth we know ahead, but for the latter, we need to account for a couple of problematic cases. In the order for linear interpolation to work correctly, we need to have samples from same depth, thus forming a square (at least when we want to avoid the hassle of dealing with four additional positions and weights). This means we need to find out the maximum depth in the proximity of our query by checking if tiles around query are in grid cell distance. If they are close enough, we proceed to consider their depth as well.

Normals. Since surface normals are directly calculated from local elevation gradient, it only makes sense to query them from the elevation map. To do calculations in 2D field of values, we need to gather maximum information from sample surroundings. That means doing 8 elevation queries forming a square around our query point. Here we have to perform full query instead of NN, as we would suffer from same issues with float precision. Knowing surrounding elevation is not enough, of course, we need to set the calculation in context of the Earth, creating a fan of vectors from center to all surrounding samples, calculating normals of adjacent ones, and averaging those into final returned normal. (see figure 3.7)

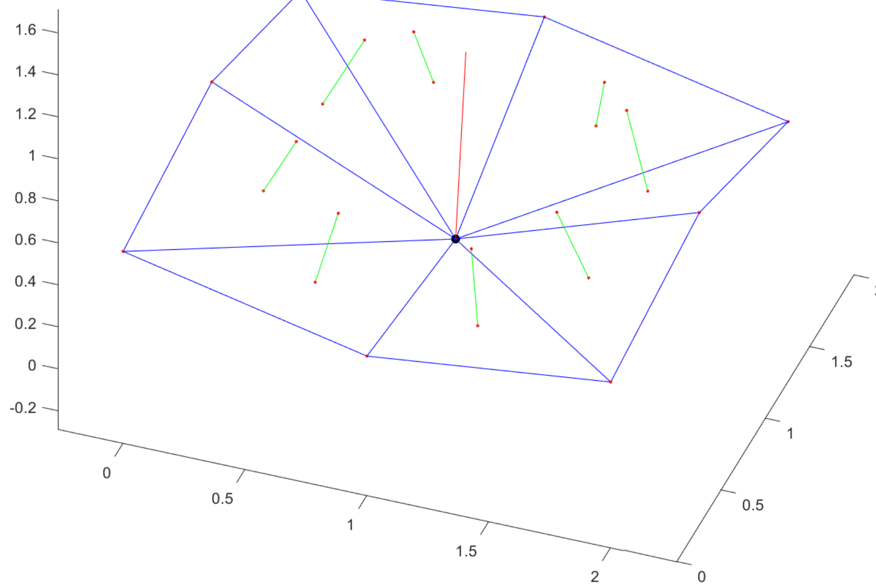


Figure 3.7: The elevation map normal query, the red dots are considered samples, the green lines are normals of triangle fan around the query sample, and the red line is final normal as average of triangle fan normals.

3.3 Imagery

This section discusses some of the available options for imagery coverage of Earth surface. Even though these tiles could contain basically any kind of 2D data, we will focus on satellite imagery, as that is the type we will display first and the most. Our first interest is the format used for particular tiles, then we will take a look at a couple of relatively accessible datasets, and in the end, dataset intended for future use and its processing will be elaborated.

3.3.1 Format

When talking about tile format, we have very few variables to consider this time around. Essentially, we can specify compression and resolution in pixels.

With compression, it is mainly the decision between lossy and lossless, as it is JPEG for the former and PNG for the latter, but there are still options to be explored later. For instance, long-time standard JPEG has a "new" contender in the form of BPG format [7], which prides itself on better quality/size ratio. There are unfortunately no Java libraries for its decoding, and even though we could run provided JavaScript decoder directly in Java interpreter, the performance would be questionable to say the least, so we will have to set it aside for the time being.

[7] https://bellard.org/bpg/

The issue we can have with lossy compression is that same compression can make a different type of compromises on neighboring tiles, and the edge between them will become more visible. This problem did not become all that visible during testing, so there is no other reason not to use JPEG.

As far as the resolution goes, our baseline is at 512×512 pixels, but based on observations of different map viewers such as *google* or *bing*, squares 256×256 seems to be quite popular. Without further measurements, we can pick the higher resolution, and decrease it at any later time when we have more usage data.

3.3.2 Sources

From available sources, we will be considering four most interesting ones. Our factors for the dataset selection are for the most part maximum resolution, global coverage, price, and quality.

Blue marble. This dataset [8] is provided by NASA as global base map constructed from countless captures to maximize visual quality as far as color consistency and cloudlessness goes. It is provided for free in three versions for each month, one with captured colors only, one with added topography (3D effect), and one with added bathymetry (oceans look more lively).

When converted into our quadtree, it is complete all the way to depth 6, making it sufficient for some large-scale scenarios, but not for all, and datasets from other sources are not very well fitting with it (figure 3.8).

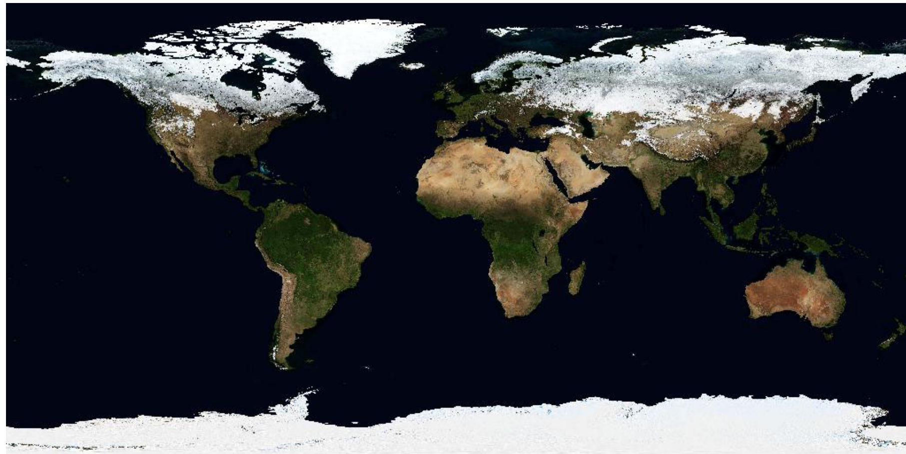


Figure 3.8: Top level of blue marble dataset.

Black marble. The only notable night dataset, more specifically its second iteration from 2016, black marble from NASA. This dataset [9] is available in

[8] https://visibleearth.nasa.gov/view_cat.php?categoryID=1484
[9] https://earthobservatory.nasa.gov/Features/NightLights/page3.php

the same resolution as Blue Marble and could be used simultaneously with its day version for 24-hour large-scale scenarios. With traffic all around the globe, and switching between night and day texturing depending on simulation time sun position as a temporal cue could be beneficial (figure 3.8).

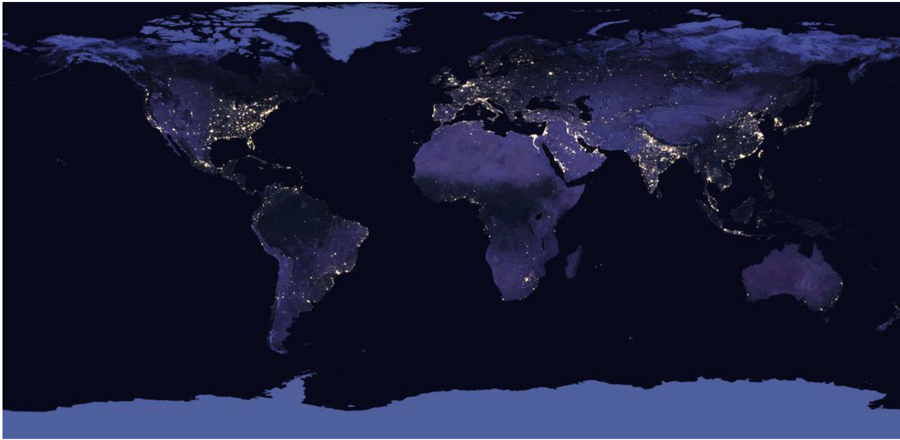


Figure 3.9: Top level of black marble dataset.

Landsat 7. This dataset is also publicly available, but unlike blue marble, it is not in the form of processed mosaic but raw satellite flyover data that contain fragments and clouds. These are then processed and merged with other datasets by private companies and sold [10] as global base maps.

This dataset is not a true color capture; instead, other bands were mapped into colors in visible spectrum: mid-infrared to red, near-infrared to green, visible green to blue. This composition gives dataset the recognizable and unfortunately unnatural pink-lime scheme.

Landsat 7 is used in the current visio, but its fidelity outside of selected part of USA is utterly insufficient, and it needs to be replaced (figure 3.10).

Sentinel 2. Sentinel dataset was a similar story as Landsat 7, at least as far as availability of its processed mosaic goes, until August 2017. The mosaic was released [11] for the public in its full resolution. Whole dataset can be viewed online [12], and downloaded from Amazon Web Service (AWS) bucket [13] (in requester pays mode — about 380USD for complete dataset of 4.2TB) (figure 3.11).

Planet SAT. This commercial dataset from Planet Observer [14] is collection of tiles from Landsat 7 and Landsat 8. Its main advantage, besides visual quality, is option to expand this 30m/sample dataset by compatible high

[10] `http://cms.mapmart.com/Products/SatelliteImagery/EarthSat.aspx`
[11] `https://eox.at/2017/08/sentinel-2-global-cloudless-mosaic/`
[12] `https://s2maps.eu/`
[13] `https://eox.at/2017/03/sentinel-2-cloudless-original-tiles-available/`
[14] `https://www.planetobserver.com/`

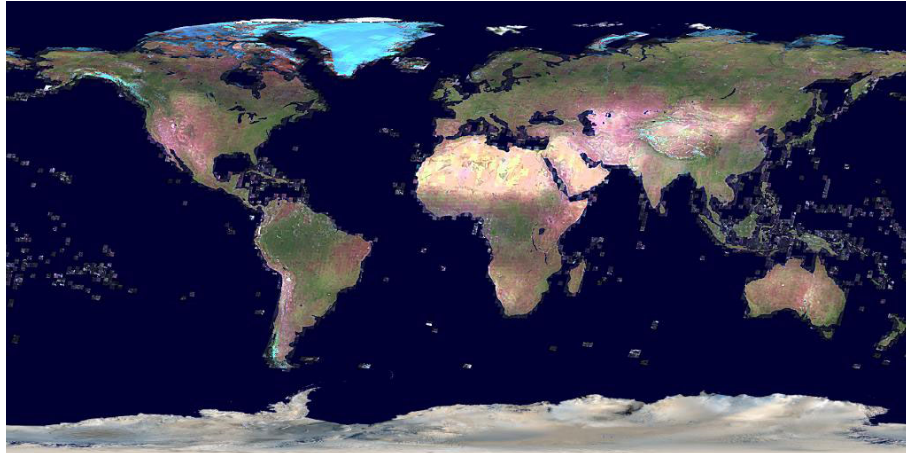


Figure 3.10: Top level of Landsat 7 dataset.

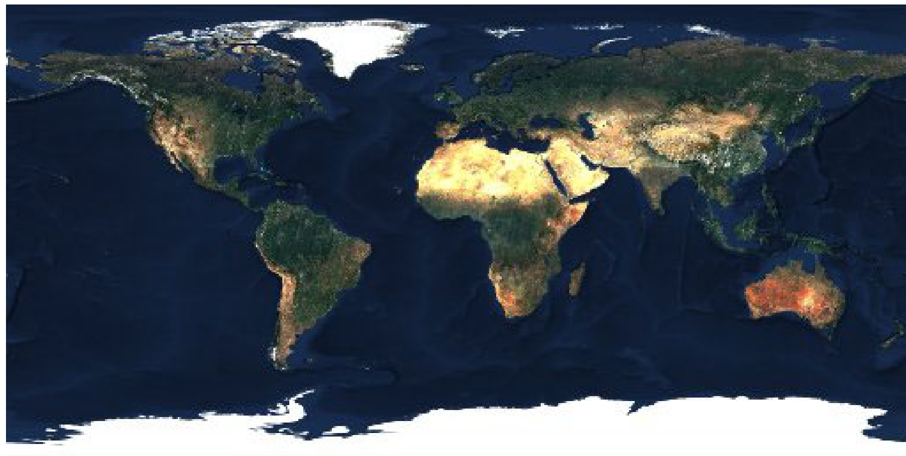


Figure 3.11: Top level of Sentinel 2 dataset.

resolution imagery (up to 0.5m/sample). The only drawback is the price, starting on 12 000 EUR for a single customer, and ending at 60 000 EUR for multiple customers (figure 3.12).

Summary. In table 3.1 is a summary of considered datasets and their properties.

3.3.3 Sentinel 2

Although we explored some other options that were not listed above (not suitable), Sentinel 2 dataset seemed like the best solution for the time being. The problem is, it is not usable out of the box, as there is no complete coverage level with fitted seas (such as in Landsat 7 case), and they need to be added in one way or the other. There are also occasional fragments of missing data and pieces of clouds that ought to be removed before usage

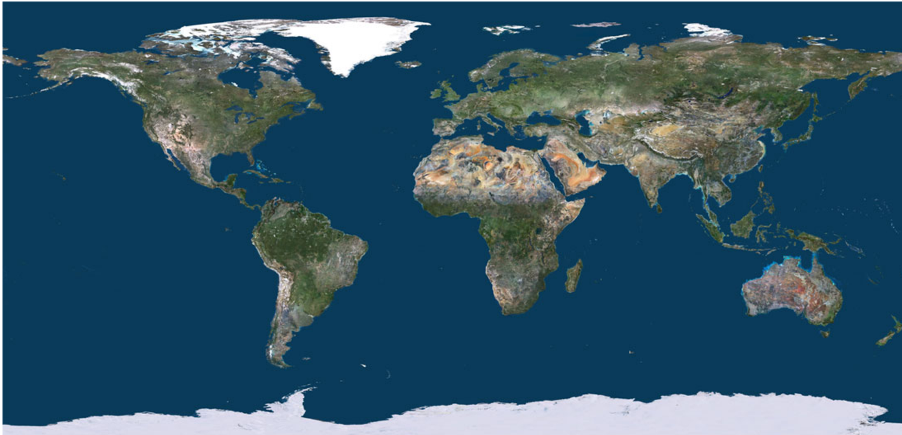


Figure 3.12: Top level of PlanetSAT dataset.

Name	Resolution	Quality	Captured	Cost
Blue Marble	500m	Consistent, complete	2004	Free
Black Marble	500m	Consistent, complete	2016	Free
Landsat 7	30m	Not true color, 5% cloud	1992-2000	1.500 USD
PlanetSAT	30m	Consistent, no sea data	2013-2017	60.000 EUR
Sentinel 2	10m	Consistent, missing data, 5% cloud	2016-2017	380 USD

Table 3.1: Comparison table for considered datasets.

(figure 3.13).

In following text will be elaborated the concept of preparation of sentinel data, not actual process that was completely implemented. Any work with the dataset this large requires a lot of time and is out of the scope of this thesis.

Figure 3.13: Sentinel 2 dataset coverage, red pixels represent available tiles.

Missing data. First in order to fix are missing samples. We can use a complete dataset to fill any missing samples one to one, but the color scheme will not match. Thus we need to find a mapping from one color space to the other based on correspondences of data we have in both datasets (figure 3.14).

This simplest solution would be to assume that the mapping does not

depend on any variable (although we can expand this model by inclusion of latitude and longitude), and follows linear equations $C_1 \cdot M + K = C_2$, where C_1 is RGB sample in complete set, C_2 is sample in incomplete set, and M with K are vectors of three constant values.

Each correspondence will thus generate three equations of six unknowns:

$$C_{1R} \cdot M_R + K_R = C_{2R}$$

$$C_{1G} \cdot M_G + K_G = C_{2G}$$

$$C_{1B} \cdot M_B + K_B = C_{2B}$$

We technically need only two correspondences, but to increase the precision, we can get as many as we like and solve resulting overdetermined system using least squares. This approach works, but only locally, as unlike with blue marble, sentinel data were taken at various times of the year.

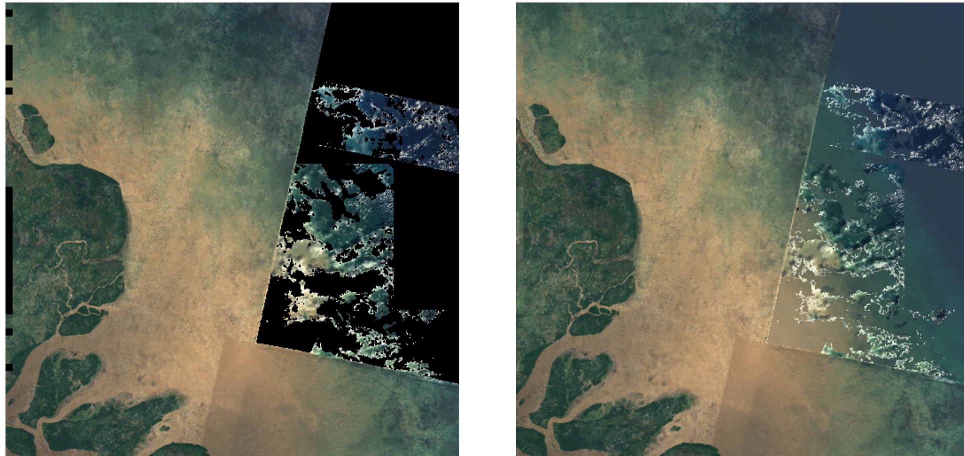


Figure 3.14: Example of a tile with filled data from blue marble with color correction.

To apply it globally, we could try using correspondence that includes latlong coordinates, or just calculate local mapping for multiple locations, and interpolate it everywhere else.

Coastline. Since sentinel dataset does not have any sea coverage besides land overlaps, we need to add the sea values from a different dataset (assuming we do not want constant value). In theory, transitions of missing data should handle this problem to a degree, but we can help it a little bit more. Using GSHHG dataset [15], we can create artificial shoreline in a fixed distance from the real one, and use it to consistently cut off missing ocean data (figure 3.15). Then we can proceed with missing data substitution.

[15] https://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html

Figure 3.15: Example of coastline cutoff as done by EOX.

Clouds removal. Even though the sentinel dataset is near cloudless, there are still patches of clouds here and there, and we could try to remove them (figure 3.16). The solution could be approached in two steps, where the second is already solved — filling in missing data. Since cloud pixels serve no real purpose, we can mark them as missing data, and fill them with something even slightly more useful.

Reliable detection of cloud data might be a bit tricky, but we can call for help to our trusty blue marble. We can try to assume that if there is an almost white pixel in sentinel data, and at the same time a dark pixel in blue marble, the pixel in question is a cloud with its near surroundings.

Of course, this problem is much more complicated and requires more attention in the future. We can use some of the multi source or hybrid gap filling/inpainting algorithms, summarized in a survey article by Desai and Ganatra [11].

3.4 Mesh

Now that we have all the necessary raw data, we can move on to the construction of actual mesh that will be displayed in the scene. This construction process is composed out of three main parts: generation of base geometry, decimation of tile insides, and generation of stitch geometry.

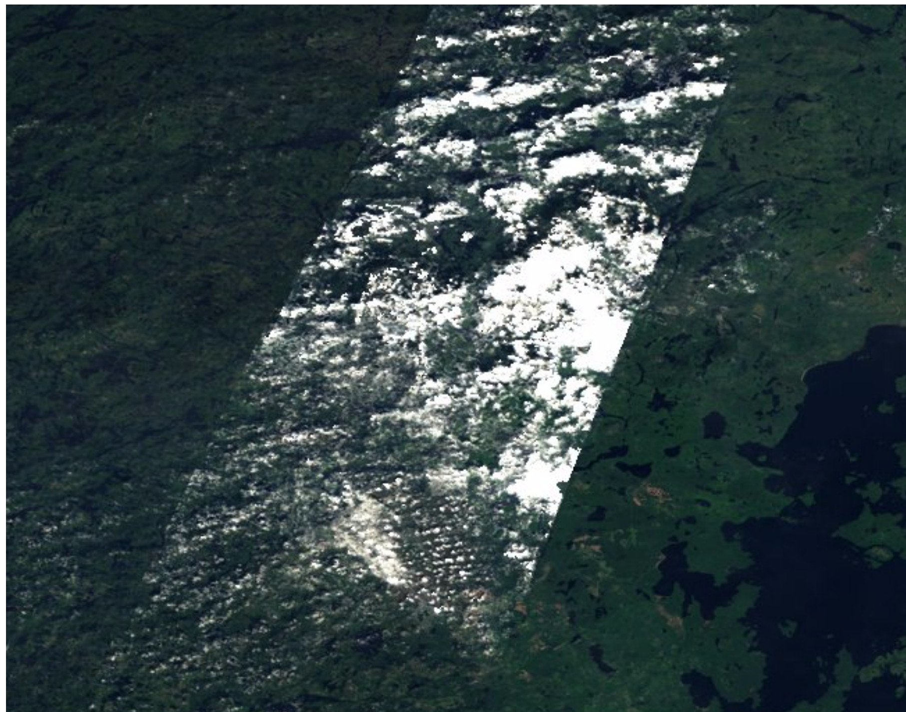


Figure 3.16: Example of more prominent cloud coverage in sentinel dataset.

3.4.1 Generation

When generating tile vertices, we need to place them correctly with respect to their corresponding elevation samples. Since those are defined for the middle of pixels in elevation data image, we need to place our vertices in the same position. That leaves tile edges — which are shared with neighbors — for stitching purposes (see figure 3.17).

We first generate all vertices in latlong coordinates and then transfer them to Cartesian while adding radius of the Earth and elevation value. At last faces are created by filling in triangles to each quad row by row. Texture coordinates are set immediately, as they are at this point same for every tile. We do not generate normals yet, as they are subject to decimation process.

When generating tiles that touch either pole, we can notice that whole bottom or top edge gets crumpled into single point of degenerated geometry, and triangles generally have ill distribution. This can be fixed by starting with single triangle at the pole and make each row towards equator one triangle longer than the last one. Since we work with square tiles, the last row will have the same length as row of its connecting "square" neighbor.

Because elevations visualized at the global scale are barely visible, we may want to exaggerate them by some factor. Of course, all graphics in the same scene needs to be elevated by equal factor. Even though it might seem as only a visual candy, it can actually be useful to have flight levels further apart

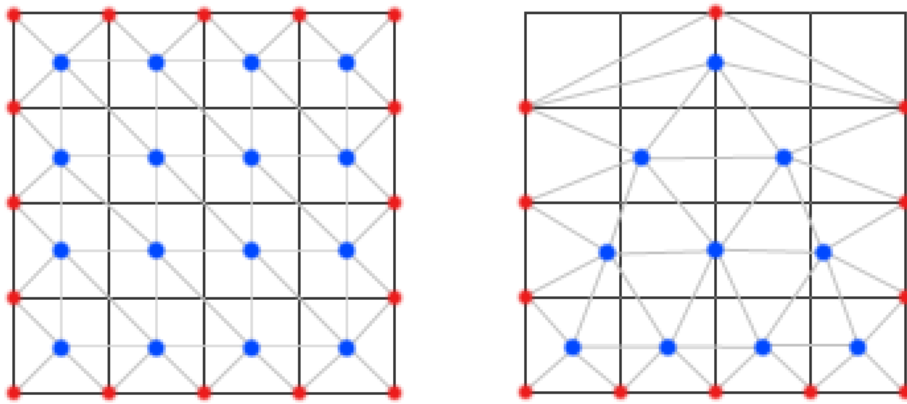


Figure 3.17: Mapping of elevation samples for square (left) and triangle (right) tiles, blue dots are dataset samples, red dots are samples from linear interpolation between tiles, grey lines are generated triangles, and black lines are areas of original samples (pixels in original texture).

(they are usually near indistinguishable at real scale), and with these needs to be elevated surface of the Earth.

3.4.2 Decimation

If we were to generate a square tile mesh at full resolution of our elevation map tile, we would get $(512 + 3) \cdot (512 + 3) \cdot 2 = 530\,450$ triangles. That is a lot for a single tile, and it is also completely unnecessary, as removal of most vertices would be hardly perceivable from intended view distance.

This means that we can dramatically reduce the number of triangles in the mesh by merely removing a number of vertices and re-triangulating holes that appear after their removal. However, why don't we just generate a mesh with lower resolution right away? Well, the explanation lies in aliasing issues. The fact a vertex is expendable is not determined by its position in the grid, but by actual elevation values of its neighbors.

For decimation purposes will be used the algorithm in listing 1. Going through the algorithm, we first load all non-border vertices into a priority queue under their plane deviance. We do not want to remove border vertices for stitching purposes. We repeatedly take a vertex from the queue and consider its removal.

In case we are above the maximum error but have not yet reached the desired number of vertices, we will keep removing. On the other hand, if we are past the maximum number of vertices in the mesh, it will not stop us from removing more if it is possible. Now onto a bit conspicuous check, that is the maximum neighbor area. This was added to the algorithm, to keep coastal sea triangles eating away vertices from land, as expanding already large triangles is not allowed due to this check.

In the last section of the algorithm is performed attempt for removal, but only in case this vertex does not have too many neighbors. Each non-border neighbor of the removed vertex is then added/updated with its new plane deviance. The algorithm terminates after a finite number of steps when the necessary amount of vertices has been removed, or none can be removed any further.

As it is integral to the algorithm, we will take a look at the calculation of plane deviance. First, we calculate average plane for vertex v with base point

$$P = \frac{\sum_{t \in T} t_a \times t_c}{\sum_{t \in T} t_a}$$

and normal

$$n = \frac{\sum_{t \in T} t_a \times t_n}{\sum_{t \in T} t_a}$$

where T is set of all triangles containing v, t_a is area of t, t_c is center of t, and t_n is normal vector of a triangle t. In other words, we do weighted average (by surface area) of triangle normals and centers in triangle fan around v. (see figure 3.18)

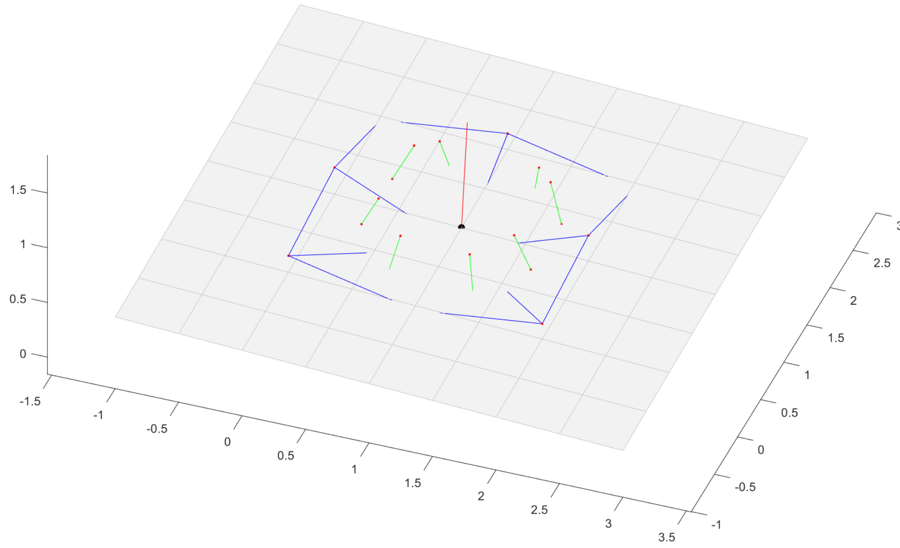


Figure 3.18: Average plane calculation, blue lines are borders of considered triangles, green lines are triangle normals in their centers, the black dot is center of mass of triangle centers, the red line is average plane normal.

Now we simply measure distance $d = |dot(n, v) - dot(n, P)|$ of vertex v from our plane to be used as deviance heuristic. The closer the point it to this plane, the lesser of an impact it will make on overall error if it is removed.

In figure 3.19 is Earth mesh wireframe render for no decimation performed at tile resolution of 128 $\times$ 128 and exaggeration of 100, tile depth 3, and in figure 3.20 the very same mesh with decimation process applied.

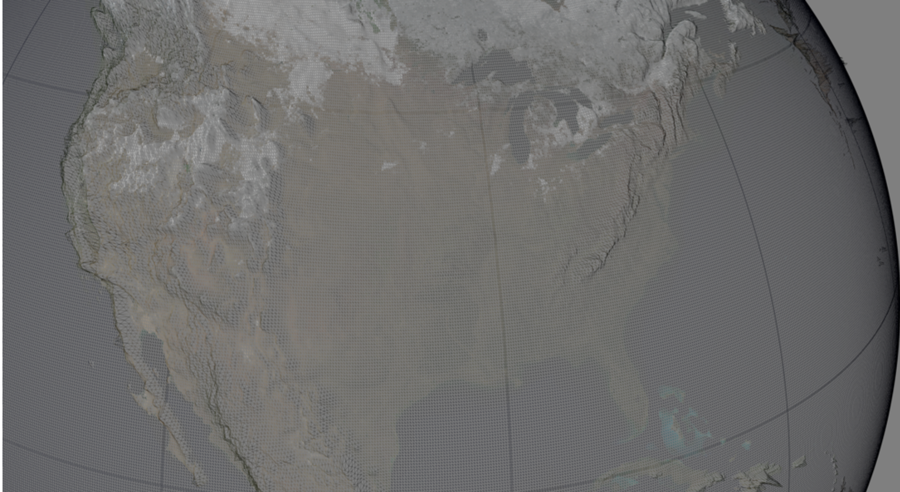


Figure 3.19: Earth surface at level 3 before mesh decimation process. (Elevation exaggeration set to 100)

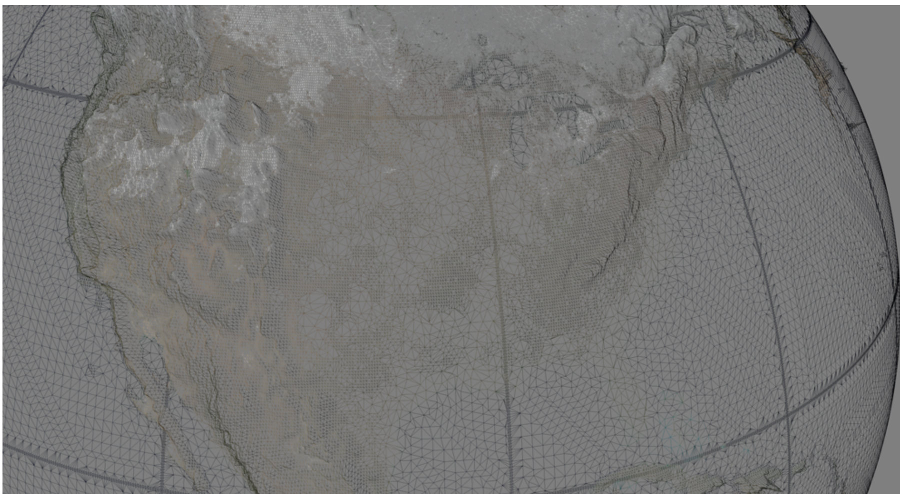


Figure 3.20: Earth surface at level 3 after mesh decimation process. (Elevation exaggeration set to 100)

3.4.3 Stitches

Let us consider for a moment, that we would not generate any stitches, and all tiles would be fully covered from initial generation. If previously defined elevation query process is used, tiles on same depth will display just fine, but once two neighboring tiles have a different depth, edge vertices may not match onto each other, and we will see holes in the surface.

There is a number of ways to deal with this problem. For example, we can extrude each tile to sides a little and downwards (towards Earth core), creating sort of interlocking wings that prevent the appearance of any holes. This solution has its own issues, as most of the others. Based on the evaluation of all these pros and cons, stitches came out as the best idea, mostly because of its perfect visual connection (although it is more demanding and complicated).

A stitch geometry (see figure 3.21) is created for each possible neighbor depth, and only the relevant one is rendered once a scene is assembled. If we choose neighboring tiles being at maximum depth difference of one, we will have to define only two dynamic stitches per tile, and each has to have only two versions (for the same and coarser level).

In this case only two small parts prove to be difficult. Tile corner and edge middle elevations are issues because these vertices can appear on tiles with depth difference of two, and thus their values must be from maximum sampling depth instead of tile depth (problem already solved by elevation map queries). The second issue is with normals, as edge normals cannot be calculated from DCEL structure, and must be queried from elevation map (again, solved by elevation map queries).

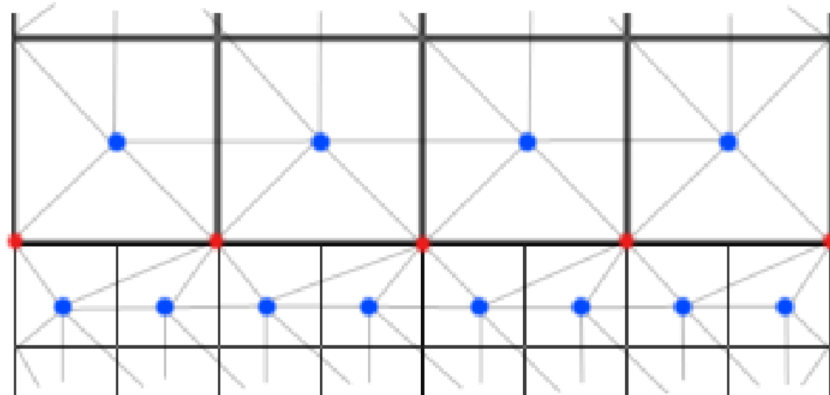


Figure 3.21: A connection of three tiles with depth difference of one. Top tile is on a lower depth that bottom tiles, all 4×4. Red dots are stitch vertices present in all tiles, blue are vertices in their respective tiles, and grey lines are triangles of all involved meshes. Stitch triangles are between blue and red vertices.

```
Data: Tile mesh as DCEL
Q = heap;
for v ∈ mesh.vertices do
    if v.isBorder then
        continue;
    end
    d = planeDeviance(v);
    Q.add(v, d);
end
while Q is not empty do
    [v, d] = Q.poll();
    if d > MAX_ERROR AND mesh.size < MAX_SIZE then
        break;
    end
    if v.maxNeighborArea > MAX_TRIANGLE_AREA then
        continue;
    end
    N = v.neighbors;
    if N.size > MAX_NEIGHBOR_COUNT OR !mesh.remove(v) then
        continue;
    end
    for n ∈ N do
        if n.isBorder then
            continue;
        end
        d = planeDeviance(n);
        if Q.contains(n) then
            Q.update(n, d);
        end
        else
            Q.add(n, d);
        end
    end
end
```

Algorithm 1: Decimation of triangle mesh

Chapter 4

Rendering Earth surface

Now that we have prepared all the necessary data, we will take a look on how to get them on the screen in the most efficient manner, as it is impossible to even to fit all of them in main memory, much less draw them. In the following sections will be elaborated the issue of spatial precision as foreshadowed in the introduction, algorithm of view based scene construction will be presented, followed by sections devoted to other optimizations that help us to put more on the screen.

4.1 Spatial precision

As we display Earth in various levels of detail and positions of the camera, we encounter issues with floating point precision near the surface, specifically when we start recognizing objects on screen that are meters apart. The moment this issue appears, of course, varies case by case, but generally, it is always eventually there. We can safely assume that users will never require higher precision than millimeters, so infinite precision is not really what we are after.

Let us take a look what is the precision problem all about in the first place. Consider an integer with a fixed number of digits, say seven. The number that can be in this integer can be for instance $i = 1234567$. Now we can use another number, call it exponent, to move decimal point to left or right, so $e = 3$ means $i = 1234567000$, or $e = -2$ means $i = 12345.67$.

Now imagine our integer is the radius of the Earth in meters $r = 6378100$. If we would want to record two positions hundred meters apart, we have no issue $r_1 = 6378100$, $r_2 = 6378200$, $d = r_2 - r_1 = 100$. However, in case we wanted a distance of half a meter, we are suddenly out of digits, and have to round up or down, creating a visible error.

Of course, the problem is not only in spatial positions relative to coordinate system center but also in precision in depth buffer during rendering. Even though we may have sufficient precision on x and y axis (the difference due

to rounding is sub-pixel), we may still encounter z-fighting. As suggested by Sellers et al. [6], we can tackle this issue by splitting frustum along the Z axis and rendering the scene in multiple passes.

There are several options how we can deal with the insufficient precision of floating point arithmetic, namely:

- Moving all calculations to double precision. This solution is straightforward to implement but does not scale very well at all. On the CPU is the problem not that critical, as double precision arithmetic must be used either way, but most GPUs have only a few units for double precision computation, and those that have sufficient amount are overly expensive.
- Maintaining transformation matrices in double precision. This allows us to have all the data used on the GPU in single precision, most importantly vertex positions, and deal with the doubles only while calculating transformation matrices. That, unfortunately, creates quite a strain on CPU side with numerous already expensive matrix multiplications slowed down by double precision. Furthermore, it indeed does not scale well for cases where not only a model transformation matrix is in double precision, but all accompanied vertex positions as well, in which case all have to be transformed on CPU.
- Creating a fixed grid of local coordinate systems. In this case, we would discretize in fixed size blocks that use single precision, but whenever requested, positions of higher precision to given point can be produced. This would mean that camera movement between these blocks triggers recalculation of all positions relative to the center of block camera is in. This can be of course done on the shader, as by definition everything within the same block is already prepared in sufficient precision, and positions from other blocks can be prepared by single MAD [1] operation per position, assuming we intend to use these positions only for rendering and thus screen space error is insignificant for objects that are distant.
- Shifting coordinate space with camera movement. This method relies on coordinate system change whenever screen space precision would be insufficient, meaning when display error reaches one pixel, coordinate space center is shifted in the current position of the camera minimizing it. In coordinate system change must be recalculated all positions at the cost of one double precision addition and cast to single precision which, unfortunately, cannot be moved to the shader. Unlike with fixed grid method where maximum error is determined by edge length of the grid, here we have to check it with every camera movement. Effectivity of this method strongly depends on the implementation of this check. For instance, if we were basing our error only on the distance of camera from coordinate center, it would take as little of CPU time as possible, but when the camera is moving by vast distances during whole Earth view,

[1] MAD operation, multiply and add, is often implemented as single instruction in hardware, exploiting nature of both instructions working from lower orders to higher. Dependent read after multiply instruction is removed, and overall precision is increased as normalization and denormalization are not performed between mult and add.

recalculation of all positions would be requested pretty much in every frame. On the other hand, if we were to calculate screen space error for every visible entity, the check would take significant chunk of time. The whole matter is quite extensively covered by Thorne [10], called floating origin.

We will take a closer look at latter three solutions.

4.1.1 Double precision matrices

This method is exceptional in its implementation simplicity since the only thing we have to do is to declare model and view matrix constructions in double precision, cast their result to float and carry on with transformation matrices as in any other visualization system.

We have to keep in mind that lights calculations, unlike with other mentioned methods, have to be done in camera coordinates instead of world coordinates. Light positions, in this case, have to be put into view space on the CPU using double precision and then be transferred to GPU to be used in a fragment shader.

The most significant disadvantage of this method is that we have no way how to deal with objects that have their vertex positions in double precision and cannot be converted into single precision. In these cases, we have to perform the transformation of those vertex positions on CPU in double precision which can prove to be quite expensive indeed.

This problem will most likely arise in case of objects that are way too large, such as flight path of an aircraft flying from Europe to the west coast of USA. In those cases, the center of mass all positions are relative to will be too far, and when looking at landing section of such flight, we would experience significant jittering.

4.1.2 Dynamic center method

As described earlier, we are shifting the coordinate system center to current camera position whenever screen error for displayed objects exceeds one pixel. Firstly, we need to take a look at requirements on object implementation side, most general use case, and then less frequent specific cases.

Since we need to be able to recalculate each position from World Coordinates (WC) to Shifted World Coordinates (SWC) and we do not want to do it in every draw call, we have to retain both at all times. Then a call with the new center in WC double precision must be accepted and propagated by those objects generating positions in SWC. These then must be automatically used in all draw calls. This update call should be made in every frame where camera position changed and screen space error is large enough. This leads us to error calculation. We stand in front of optimization, where we want

to minimize the amount of work taken by error recalculation, and also the number of SWC recalculations.

Possible approaches are:

- Recalculation is done at a fixed distance of the camera from SWC origin. This means that error will never exceed a specific value, but also that when the camera is moving fast through empty space, error on no object is over one pixel, yet recalculation of SWC is done in every frame.
- When a position enters a radius around the camera, which grows larger the further camera gets from SWC origin. Here we know recalculation of SWC is done only if it is necessary, but at the cost of additional distance computation to all positions with every camera movement. Since we need to take into consideration the worst case scenario, we may have to resolve to consider bounding spheres during this check. We can employ acceleration structure to search through all present positions faster and avoid those that are apparently out of range.
- Recalculation is done at a dynamic distance of the camera from SWC. In this case can be exploited the fact that we know the camera is mostly moving around the Earth observing airplanes that are moving in limited space above the ground. This way we can change the recalculation distance based on distance to the Earth surface. This approach is much less flexible but offers close to optimal results for next to none computational cost.

In the ideal case, the fast third option would be implemented alongside with the second one just to keep the system ready for general cases.

Now we will take a look at the most general use case of the system described in the graph below. We can see there that model data are given in single precision while model and camera positions are in double precision (as they will be in all following cases). Here we simply apply SWC origin position in WC to view and model transformation matrices and continue using them in rendering pipeline (figure 4.1). This case is from a computational cost perspective the cheapest, as we do not have to touch the model data at all. As an example, we can take a look at any airplane model that is placed somewhere in the world.

A case that proves to be a bit more complicated is when we have model data in double precision and thus raises a question how to accurately represent it. If the data perchance do not lose precision by conversion to float, we can do that and move to the previously presented case. Another option is to check whether the center of mass for all positions within our objects are close to each other enough to not generate any fragments, and if they are, we can recalculate them as relative to the center of mass, and apply its offset on model transformation. Even though this might seem like the general case again, it is not. We need to keep in mind that whenever model transformation changes, this offset needs to be kept separate and reapplied

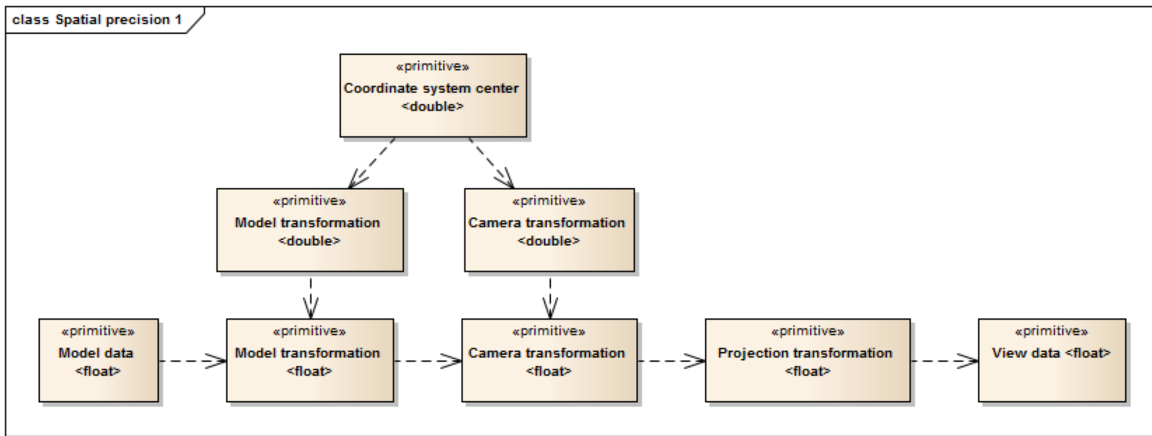


Figure 4.1: Data transformation with input in single precision.

(figure 4.2). As an example of this case could serve Earth tile mesh generated at runtime in double precision relative to Earth center, that is small enough to be recalculated.

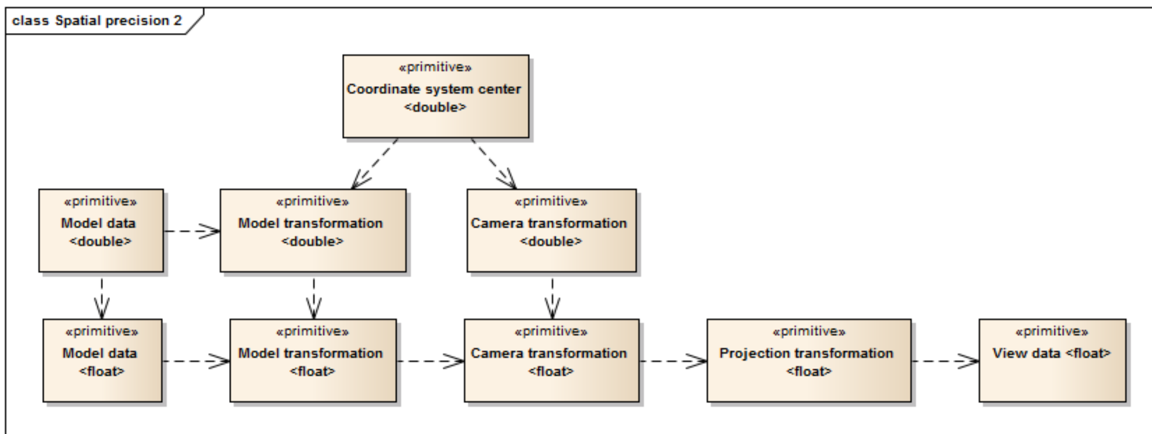


Figure 4.2: Data transformation with input in double precision with possible offset.

The third case is essentially the previous one without the preprocessing option. Here we have to recalculate all positions of the mesh relative to SWC origin with its every change (figure 4.3). Cases like these should be rare, mostly occurring for Earth tiles that are too large. It is important to note that while Earth surface is dynamically generated, and tiles that are covered by this cases do not stick around for long. If we are looking at such tile from proximity allowing us to see some errors, it means this tile is not detailed enough for our view, and it will be replaced by different one soon enough, possibly one covered by the previous case.

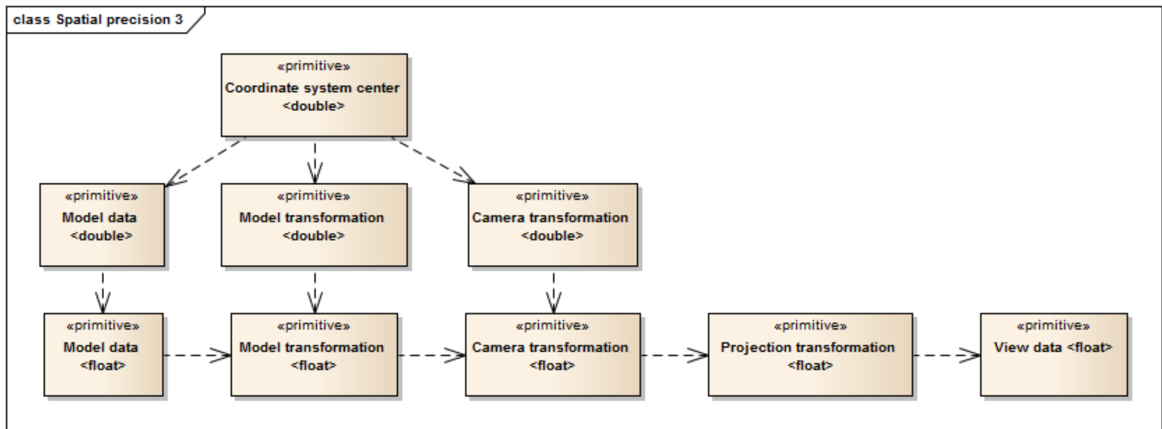


Figure 4.3: Data transformation with input in double precision.

4.1.3 Fixed grid center

In this method a fixed grid is used as a substitute for higher order decimals. Given three integer address of a block in 3D space, and then three single precision float position within that block, we can calculate WC position by multiplying address by block edge length and adding block position.

Implementation of this approach requires that every position has an address of block it belongs to in addition to a single precision position within the block. Position is then recalculated every time camera moves using formula

$$x = -cp + (va - ca) \cdot e + vp$$

where x is position in camera space, cp is camera position in its block, ca is camera address, va is vertex address, e is block edge length, and vp is block position of our vertex. This formula uses integer and single precision float arithmetic only, and we can see that the smaller difference in addresses is, the smaller the error is (figure 4.4).

When both the camera and the examined vertex are in the same block, the error is minimal. This minimum error is then determined by block edge length. We can derive desired edge length by simply taking desired precision in world units, which would be at least centimeters, and calculate how many of those fit into six decimal places of precision for single precision float. By this logic, we arrive at edge length $10^6 \cdot 0.01 = 10^4 m$. Considering our use case needs to fit in the Earth of $6300km$ radius with the addition of $200km$ above the ground, we divide that number by $10km$, we will arrive at the grid of $1300 \times 1300 \times 1300$ virtual blocks.

Recalculation of positions in this method is as opposed to dynamic center method arguably more demanding, but it can be done in the shader per vertex, which is much better, as we want to spare the CPU as much as possible. That means we need to pass to shader vector of three integers for

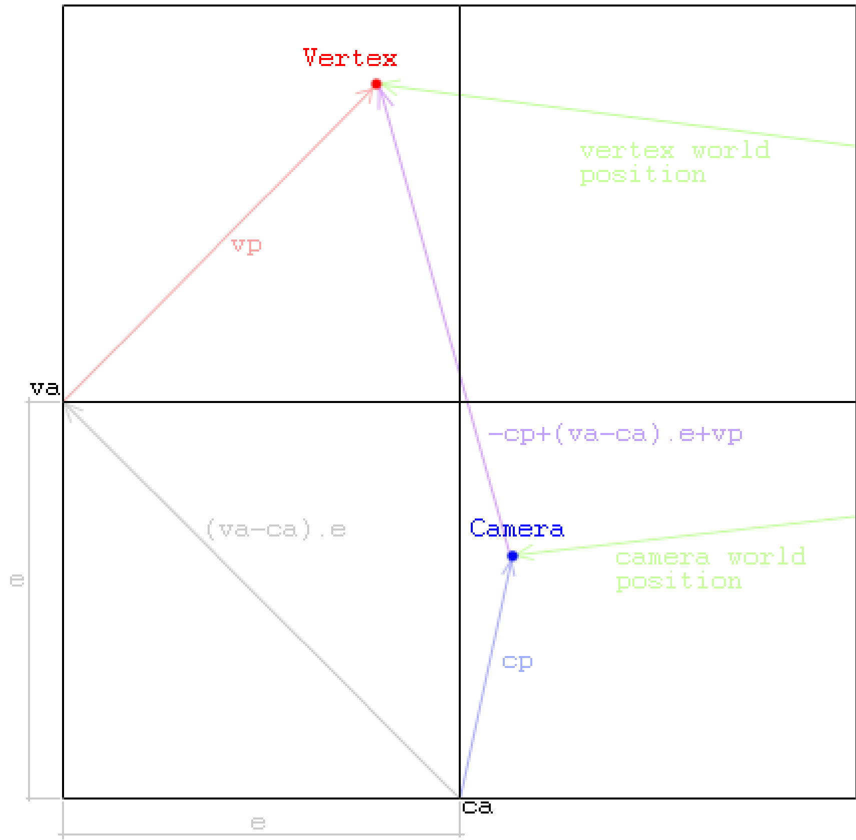


Figure 4.4: Vector relations in fixed grid approach. A camera in a block with address *ca* and offset *cp* is looking at a vertex in a block with address *pa* and offset *vp*. Resulting vector to the vertex in camera space is in purple. World coordinate space origin is out of the frame, with green vectors hinting its (far) position.

cases where vertex positions are offset only by the transformation matrix, or array of three integer vectors as vertex attributes in case these vertices are defined in double precision.

Coordinate processing cases are similar to those with the dynamic center method with the only difference being modification primitive stating block address instead of coordinate system center.

4.1.4 The selected option

Considering all advantages and disadvantages, *fixed grid center* seems like the most flexible option, but it is quite complicated as far as implementation goes. We can use the *double precision matrices* method for the time being with little to no intrusion to our system, and consider changing it later on when it seems unavoidable.

This leaves us one case that is likely to occur, causing precision issues and is not handled well in our solution of choice. Whenever we receive an object that is too large that even relative position to its center of mass produces rounding errors, we can split it into multiple smaller parts that do not pose such problems. We could, of course, place this responsibility on users of the system, but since we are trying to minimize necessary knowledge for system use, this is the next best thing.

4.2 Slice construction

In this section, we will be elaborate on the process of Earth scene construction. It is, in essence, a slice through our quadtree at different levels, ensuring each point on Earth surface is represented exactly once in the scene. We will first take a look at what tiles are desirable in the scene with respect to specific camera configuration, and then construction algorithm will be elaborated on.

Topics lightly touched in the algorithm overview will be properly described, such as heuristic of the desirability of tiles, enforcement of consistency, and stitch switching connection. We will also briefly mention used methods for reducing of the number of rendered tiles based on the view frustum.

4.2.1 Visual fidelity model

To better understand how to build a visually appealing scene, we need to formalize what visual fidelity is actually desirable and how to quantify it.

Screen texture coverage How many texels of used texture correspond to each pixel of rendered surfaces on the screen. If there are too many texels per one pixel, resulting image will show signs of aliasing. This manifests in randomly disappearing details for instance. The other extreme, one texel per too many pixels, will lead to an equally undesirable issue, that is lack of detail and overall blurriness. Since application of mipmaps can mitigate issues with oversampling, we really care only about closest available texture where maximum pixel/texel ratio is below one.

Maximum elevation deviance What is the maximum error of Earth tile points between a maximum and used fidelity in screen space? In other words, how visible is the difference on the screen? This value can be calculated from precomputed error in world units for each level of detail and the corresponding size of world unit in pixels. If on given tile is the elevation error 50 meters or less, and one meter is at the closest point on that tile 0.001 pixels, we can conclude this error will not be visible and thus it is acceptable. We want this error as close to one pixel as possible not to waste performance.

Consistency What is the maximum difference between different tiles displayed on the screen? It is a crucial visual aspect to keep the graphical

fidelity on a similar level for all elements on the screen, as the gap only highlights imperfections of low fidelity ones. This can be expressed by placement of above attributes on an exponential curve when calculating priority of what to place in the scene next.

4.2.2 Algorithm base

In this section, the currently implemented basic algorithm will be described and its issues will be discussed. A short outline can be found in listing 2.

At the beginning is created a priority queue, where objects (quadtree tiles) are sorted by their current desirability with respect to the current camera view. We add root tile and move to the loop of perpetual improvement.

A tile is examined in the loop for whether it is already planned in the next cut, as it might have been inserted during application of neighboring tile. Then whether it is sufficient for the current view, which tests its visibility (all culled tiles are considered sufficient) and its desirability. Then we make sure the tile and all its connections are ready (more on that in subsection 4.2.5), and if they are not, a prefetch of their data will be scheduled. Only if a tile is not planned, sufficient, and ready, it will be added to the next cut. In the end, all tiles that changed their state are either activated or deactivated, which essentially means addition or removal from the rendered scene, respectively.

This approach is reasonably simple, but it bears a burden of low scalability for a couple of reasons.

- Unnecessary loads of tiles that are not sufficient will be performed. We can see from the algorithm, that the cut moves from root to leaves and all tiles in between have to be loaded, even though we may know ahead they are not sufficient. This progression makes sense at first cut search, but when a camera moves from one cut in greater depth to another one, it is no longer necessary.
- We have to traverse whole quadtree up to our cut in each frame. This does not seem to have an impact at the moment, but it may have it when we start exploring datasets with a depth greater than ten.
- Assumes everything in the tree above the cut fits in memory. In case we run out of GPU memory during exploration, it may happen that root node will be released along with unwanted nodes, and Earth will change its fidelity to its lowest level for a couple of frames. It is important to note that this problem can also be solved by adjustments to data release policies.

4.2.3 Algorithm extension

The algorithm after extension is very similar to the base one, but it has two important changes to address highlighted issues.

The first change is the addition of progression stack to our original probing mechanism. Instead of checking whether a tile is ready for being in the scene, we ignore its state and along with its addition to cut add it to progression stack. Each element of this stack also holds information on min/max tile quality in the cut at that time and number of tiles that are not ready. Once we traverse to the desired cut, we find a position in probing time that offers the *best missing tiles* to *gained quality* ratio, and prefetch those tiles. Then we find the best ready cut and activate those tiles.

This change will behave nearly identically to initial state in the base algorithm, but after a deep cut has been established and changed, we suddenly no longer need all tiles above the cut. We can always display previous one (the one ready cut), and move with each frame a little closer to the desired cut. This change removes not only issue with unnecessary loads, but also problem with having to keep everything in the memory.

The second change is a rather simple one; we initiate the probing phase at previous cut instead of root. This is, of course, simple change only as far as the logic of the algorithm goes, as we need to maintain a consistent neighborhood, and thus each tile not only needs to keep an eye on lower depth nodes when improving but also on higher depth tiles when degrading. This way we no longer have to traverse the whole quadtree, and thus the number of steps on probing stack is significantly smaller.

4.2.4 Desirability heuristic

When mentioning a tile quality in construction algorithm, we never talk about the exact properties or ways it is calculated, and for a very good reason. The inner working of the algorithm should not depend on tile fidelity evaluation, and the value itself should represent quality only in two ways, sufficiency and relative quality between two tiles.

As far as sufficiency goes, we want to have a known boundary, that is crossed once the camera is far enough. As described in visual fidelity model, we do care mostly about texel density and maximum vertex displacement, but these two values are expensive to acquire, so it is not a bad idea to look for substitutes.

For vertex displacement, we can look no further than at a ratio between volume diagonal, and distance from that bounding volume. This will come out as a number around one (our boundary) that we can tune to get the popping effect at its minimum for most tiles.

Texel density is a relation between screen resolution and texture resolution, which goes to 1 if these two match.

Combining these two approximations, we want such an effect that in case of double the resolution of the screen to a texture, we have to be twice as far. Final quality formula can then look like

$$dh = \frac{l^2/\frac{h}{t}}{d^2}$$

where l is the distance from the camera to the closest point on bounding volume of the tile, d is bounding volume diagonal, h is screen height in pixels, and t is texture edge length in pixels. This heuristic is, of course, not ideal, and we can try any number of different ones, visually confirming their effectivity.

4.2.5 Adjacency enforcement

As mentioned before, we need to maintain some amount of consistency in neighboring tiles, but not only for visual fidelity itself but also because of stitching requirements. At data preparation stage, we chose a number of stitches we want to include in each mesh. This number has to be at least one, but it can go as high as we want.

We need to make sure that when two tiles appear next to each other in the scene, we have a stitch to prevent any holes between them, and since each tile has prepared stitch up to certain lower depth, we need to enforce maximum difference in depth of neighboring tiles.

When a tile is considered for improvement in our base algorithm, we need to check whether it is not about to violate restriction with any of its neighbors, and if so, such neighbor needs to be improved along with it (and so on recursively). This rule can be generalized into two connections for each tile to its upper non-parent edge neighbor at a depth lower than enforced distance.

Following on reference from the advanced algorithm, we need to deal with connections from both sides, if we ever want to perform simplifications along improvement operations. In simplification case, a different set of connections must be established in the other direction in the same manner, except this time, we will have four connections for depth difference of one, eight for difference of two and so on.

Why would we want to have higher than one of depth difference though? Since we are enforcing that a complete cut has to be loaded at all times, even when its parts are not in the view, we want to be able to emerge from finer levels as fast as possible. This holds true especially when we are inspecting airports, and most surrounding tiles are not even remotely visible.

When talking about adjacency connections, we might as well mention stitch connections. These are very similar to adjacency once, except in case enforced depth difference is greater than one, we have a connection to each lower depth neighbor instead of just the most distant one. A tile in the scene then checks if any of its connections is also in the scene, and if so, it will apply a stitch corresponding to that connection. If no connection is active, default (same depth) stitch is chosen, and it is assumed neighbors will deal with the stitch on their side.

4.2.6 View frustum culling

VFC is the process of cutting objects from rendering pipeline based on their visibility in view frustum. Using geometry of culled object for the test would be quite inefficient, and that is why we usually use substitute objects, bounding volumes. These volumes enclose processed object with varying tightness and have also varying cost of the frustum intersection test.

For instance, the bounding sphere has a fast test, but will not be very tight in the majority of cases, whereas complete convex hull of an object will have expensive test while being one of the tightest generic bounding volumes. Whole volume selection is then optimization process, where we minimize test cost and spare silhouette surface at the same time.

In our case, Oriented Bounding Box (OBB) seems like a good idea, as most deeper tiles do have square-ish shape with just different orientation around the globe. Constructing tight OBB is not as simple as it may sound, as we need to choose a correct center, rotation of its orthogonal system, and dimensions. Size is technically directly dependent on remaining two, but it is still a lot of freedom to make bad choices.

In figure 4.5 are visualized oriented bounding boxes calculated as described below. We can see there is room for improvement as these boxes can be tilted and thinned, but it is a consistent start. If we take a look at figure 4.6, we will find out that this quality issue is much less prominent on the surface generated with no exaggeration. The construction of our tile OBB will be split into three parts, one for each property.

Axis. While calculating axis for tile OBB, we start with the easiest one. Axis A_3 is calculated as a normalized vector pointing to the center of the tile. Then we calculate a help vector A_1', that leads from center of the tile to its east side. Second axis is perpendicular to third and our help vector $A_2 = A_1' \times A_3$, while first axis is perpendicular to second and third, $A_1 = A_3 \times A_2$.

Dimensions. Having an orthogonal system, we now project all contained vertices onto its vectors, and find six points defining a minimum and maximum extent. This can be done for instance by initializing all extent points to the first vertex, and then iteratively calculating dot product between a checked axis and vector from tested point to previous maximum/minimum. Consider $minA_1$ being current minimum extent point along axis A_1, and D vector from $minA_1$ to tested point P. If $D.dot(A_1) < 0$, then tested point is new minimum extent in that axis.

Having all six extent points, we can now measure a distance between them along their respective axis and get our dimensions. The measurement is done by a projection of vector between minimum and maximum onto their normalized axis (dot product).

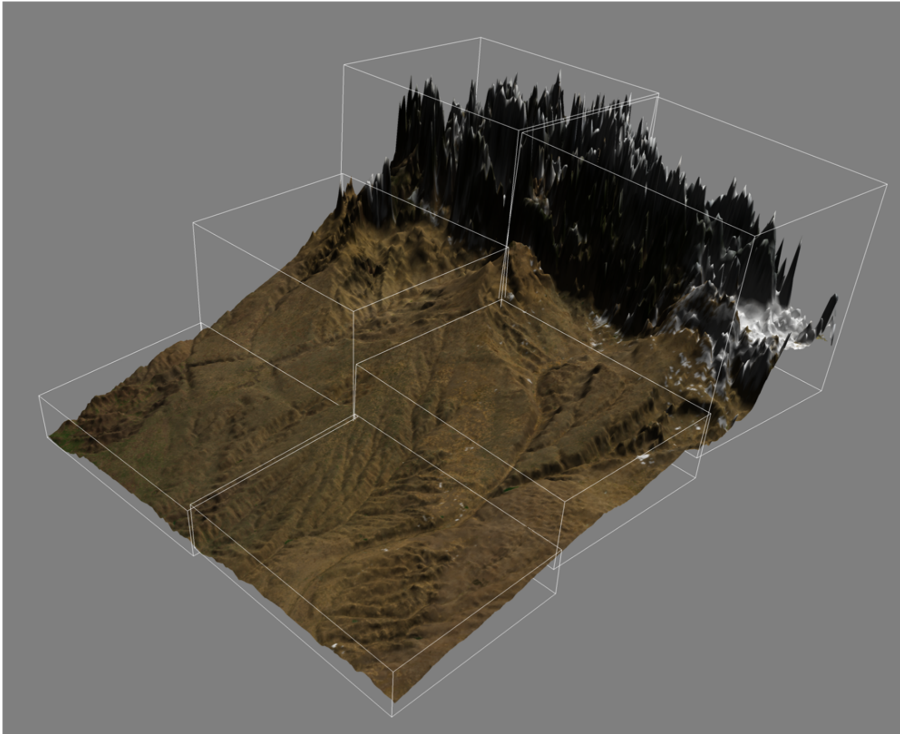


Figure 4.5: Oriented Bounding Box volumes visualized on top of Earth tiles with exaggeration equal to 100.

Center. Finding the center seems very simple, but it still takes a few more calculations. We know that center lies on dividing planes for all axis. We can construct all three planes from midpoints between minimums and maximums combined with their respective axis. Performing an intersection of planes P_2 and P_1 should yield line along axis A_3. Now we perform an intersection of the line L_{12} and plane $P3$, which is the same as the projection of A_3 midpoint onto A_3, and also same as OBB center.

4.2.7 Occlusion culling

VFC is not nearly sufficient for removal of all unnecessary tiles, as many tiles on the other side of the globe are still in the view frustum, but occluded by the Earth itself. To get rid of off all these tiles, we need to define in a simple test when they stop being visible due to Earth being in the way. Ideally, we would like to test camera position on a volume tied to culled tile, and if the camera is in this volume, tile contents cannot be visible.

To cover all rays coming from the camera towards our tile, we can put a cone on top of it. The first variable, the cone will share its axis with the center of the tile. The second variable, the cone will be tangent with Earth surface. And the third variable, we need to choose how far will be the tip of the cone from Earth surface. We need it exactly so far that every vertex of

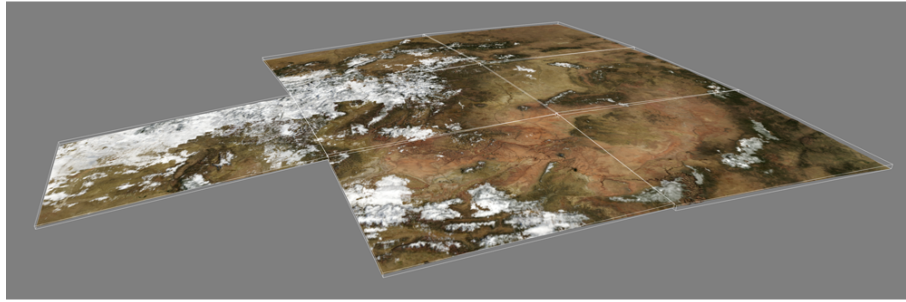


Figure 4.6: Oriented Bounding Box volumes visualized on top of Earth tiles with no exaggeration.

the tile is inside of that cone. If we place the highest elevation in the corner of the tile, we will have a furthermost possible point on tile surface from its center, and this point lies on the surface of the cone we are looking for.

Now whenever the camera is inside of this cone, tile is not visible. However, that does not hold for the volume between the tip and the tangent circle on Earth, and this area is covered rather by the Earth itself. To solve this problem, we will offset the tip of the cone to the center of the Earth, and use Minkowski sum using Earth sphere on it. This way, we will have a cone with a rounded tip with the exact size of the Earth.

To calculate it we simply need to follow angles as depicted in figure 4.7. We know vector C pointing into the center of the tile, and point P as a vector pointing to the non-pole corner of the tile prolonged to maximum elevation. Since T is lying on a tangent to Earth, angle PTO must be a right angle, which gets us using the law of sines to angle φ. We can calculate angle β from vectors P and C, and thus vector ω. Now we know that $\varphi + \omega + \delta = 180°$, we can derive δ and at last θ, which is also half our cone opening angle. The direction of the cone is reversed normalized vector C.

The test for a point being inside of this volume is quite fast. Looking at scheme in figure 4.8, we first test whether our point is in cone C_1, areas $A_{1 \dots 4}$. Then we take backward facing cone C_2 with the tip in Earth center (has the same intersection with Earth as C_1), and check whether our point is inside. If not, it must be inside area A_3 or A_4; otherwise, we verify if it is in Earth sphere and thus A_2.

4.3 Load balancing

In this section, we will discuss settings that are tied to different hardware limitations, and to the implementation of the system. First, we will take a look at data transfer overview for biggest "hitters", then various fidelity settings will be examined along with their settings mapping, and in the end, we will take a look at memory consumption issues.

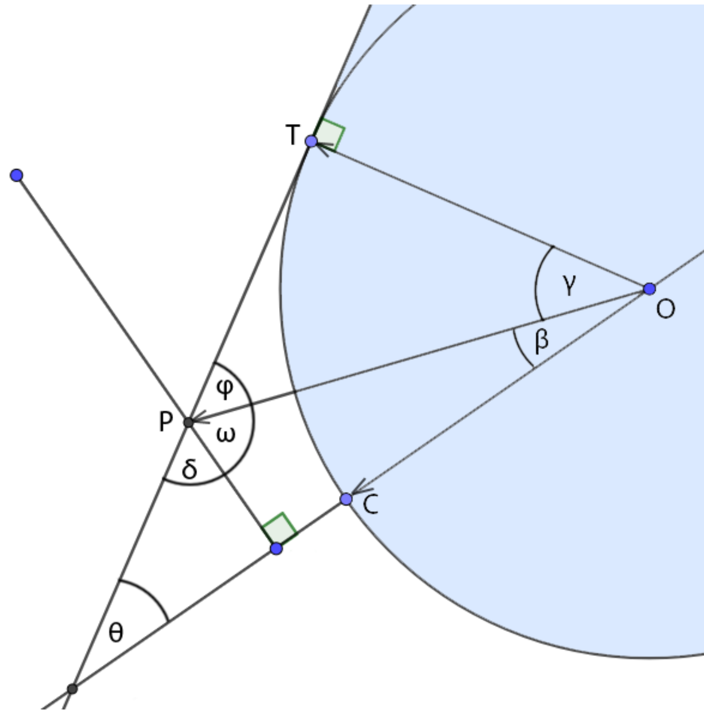


Figure 4.7: Calculation of opening angle for tile occlusion cone.

4.3.1 Data transfer

Now we need to take a look at things that do change between frames and cannot be loaded ahead of time. For instance, entity models can usually be loaded at startup of the application, as we know exactly which pieces may or will appear during the tested scenario. In our main scenario with Earth, only Earth tiles, aircraft positions, and aircraft labels will produce notable strain.

Entities. At peak times, there are 4000 airplanes changing their position every simulation tick. That means in case of instanced rendering a rendering call for every entity type in the system, which we have about 20. Of course, each entity type is consisting of multiple parts that wary based on fidelity and LOD level, but the number of calls for all should not exceed 100. Each entity needs a model-view matrix, normal matrix, and model-view-projection matrix for rendering with lighting model. That means three 4×4 matrices totaling to $4000 \cdot (4 \cdot 4 \cdot 4 \cdot 3) = 768kB$, which we can for safety round up to $1MB$ of data.

Labels. Next on the list are labels that can be handled in various ways to minimize performance demands or amount of transferred data.

- One of them is a mesh aggregating method that will squeeze all labels into a single call. This is done by a complete reconstruction of single label mesh of quads with mapping into a font texture. That means

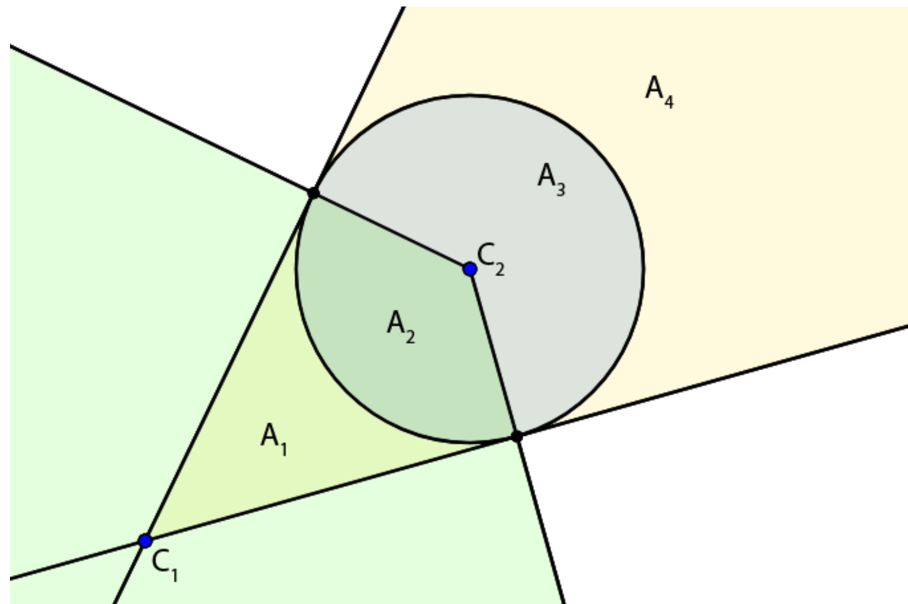


Figure 4.8: Help volumes used for point intersection test with cone occlusion volume.

every time a label position is changed the whole buffer needs to be updated. Considering average ten characters per label, four vertices per character and each vertex having UV coordinate and position, we arrive at $10 \cdot 4 \cdot (2 \cdot 4 + 3 \cdot 4) = 800B$.

Now each aircraft may have several such labels, usually no more than 4. Final number for one update is then $800 \cdot 4 \cdot 4000 = 12MB$. This is, of course, an extreme case, as not much would be visible, and culling methods are usually employed to reduce this amount, but we should still keep this number in mind.

- Another option is the use of a custom shader that will construct all the geometry for all labels on the fly from an array of characters and translation vector. For programming convenience, we will fix length of every label to a required maximum; let us say in our case 20 characters. This way, we will be sending into our shader with instanced rendering on a topology of 20 quads once for all labels, an array of characters $N \cdot 20$, and an array of translation vectors of size $N \cdot 12$, where N is the number of labels.

  Now we initiate instanced rendering for our quads, setting up instance data stride of length 20 in our text array, and instance data stride of length 12 in our translation array. This means we will be able to render all labels in a single call without constructing geometry on the CPU. In the shader, we will then use a built-in variable for vertex ID to pair a character with its quad, and thus correctly pick its offset and texture mapping. This method is, of course, subject to further testing, as in current proposition, we have to iterate through all preceding quads and their offset to get offset for the i-th quad. Data-wise, this approach

will require $20 + 12 = 32B$ per label, $32 \cdot 4000 = 512kB$ for expected maximum load, making it rather negligible.

Tiles. Lastly, camera position can change between two frames in such a manner, that the whole Earth has to be reconstructed basically from scratch (for instance changing the view to an aircraft on the other hemisphere). We have currently Earth surface textures in the format $512 \cdot 512 \cdot 3$ RGB model, leaving us with $768kB$ per quadtree tile.

Overall. Summed requirements for largest items form about $20MB$ of data per frame:

- $768kB$ for entity positions.
- $768kB$ for single surface tile, eight tiles for full HD full-screen coverage is $7MB$.
- Either $12MB$ or $512kB$ for labels, depending on the used method.
- Tile mesh data size depends on tile elevation complexity, but usually does not exceed $100kB$, with mentioned eight for full HD at $800kB$.

Discussion. The minimum expected communication speed on the bus to the graphics card is $2GB/s$, which is about PCI-e 2.0 $4\times$, technology from 2007. If we assume on-screen data change in the worst possible way, we need to allocate that bandwidth from the beginning of that particular frame.

We want 60 frames per second with $2GB/s$ at our disposal (about $(2 \cdot 1024)/60 \approx 34MB$) within that particular frame, while needing about $20MB/s$.

Of course, as mentioned earlier, we still need to wait for some drawing to be done after data transfer, but we have clear sectioning between demanding tasks in rendering, and we can pipeline rendering with data transfer in a way that does maximize overall bandwidth. Call for entity model rendering requires the least amount, so it can go first, while we are waiting for Earth textures to upload, then we can render Earth while constructing and uploading aggregated labels. Also, there are quite a few more static objects that need to be rendered, and with which we can fill up the downtime.

4.3.2 Fidelity settings

Presented slice construction algorithms do partly take care of fidelity balancing, as if there are no resources to load new improvements, the scene will stay as constructed at the edge of desired FPS. The problem arises when the amount of available power is decreased, due to a new instance of visualization being turned on for example. For these reasons, we need to react to resource availability change by redefining what is desired fidelity.

For this purpose, we can imagine imaginary slider, a number $q \in [0, 1]$, where $q = 0$ is the bare minimum for visualization of all the necessary data,

and $q = 1$ is state in which any improvement makes no or insignificant difference. We can now map any fidelity settings on this range, and whenever there is a dip in performance, we can easily move current quality factor up or down to maintain steady FPS.

Slice construction constant As mentioned before, we have quite a simple desirability heuristic for tile sufficiency evaluation. This means we can add to it an arbitrary factor from range $[min, 1]$, where $0 < min < 1$ is our bare minimum fidelity, visually confirmed. This factor can be linearly mapped onto quality slider.

Resolution multiplier Using rendering into higher resolution buffer can be easily achieved the simplest version of anti-aliasing. Since we are using a buffer of arbitrary size and then downscaling it with linear interpolation, we can define relative size as a number in the range $[0.5, 2]$.

Anisotropic filtering Even though performance impact of this option is up for discussion, it is a number we can change based on system load, and it is objectively faster setting it to lower value.

MipMaps Generation of mipmaps is rather on/off option, where we define a point on our spectrum when it is no longer being used. Turning off mipmap generation will mostly cut down necessary texture setup time, and overall demands on GPU memory (although we save only about 1/4).

Texture resolution The smaller textures we use, the more likely it is processed texels are in fast GPU memory, so it makes sense to decrease texture resolution on GPU if we are having performance issues. Since textures used in our system will be mostly squares in power of two, we can map texture downscaling to a tolerable minimum from the original size.

Lighting model Lighting model also offers only a few meaningful options. Essentially, we can turn off the specular component, and in subsequent reduction keep only ambient light. This change should be mostly cosmetic, as light should not carry any visualization data in any scenario. Cutting of lights means substantial simplification of shader code and thus higher fill rates.

It is important to note that we might want to fixate rendering quality on maximum for purposes of video recording. Low framerates are usually quite inconvenient, but when capturing 24-hour runtime, the final video gets condensed and framerate from the time of recording does not matter at all.

4.3.3 Memory limits

Because we are usually working with large datasets, memory management needs to be discussed. As our simulation machines have between 32GB and 64GB or RAM, CPU memory is not as important as it would be in different

systems. The rule of thumb is not to hold onto unnecessary data, and we should be fine for the most part.

The main issue with CPU memory is not so much the amount of space, rather than amount of generated objects. Assuming new system will be implemented on a platform with garbage collection, we need to make sure our rendering and scene construction process produces minimum amount of temporaty objects that prolong garbage collection pauses. This particular problem will be discussed in following chapter.

What can become an important issue is GPU memory. Since the transition of data from CPU to GPU memory is expensive, we do not discard objects on the graphics card as soon as they are not used, but instead, we keep them in hopes they will be needed again. Since running visio is, for the most part, the only application consuming GPU memory, we can afford to splurge it at our whim.

We do need to monitor how much memory we have left, and once we reach a certain point, we have to start releasing data from it to make room for newcomers. Of course, we can perform this release at random, but implementing specific policy may improve our performance significantly (cache release policies are quite extensively researched topic [2]).

When releasing data, we need to keep in mind that this process takes time same way any other action on the GPU does. This means that we need to release data gradually every frame to spread out the load, so we do not experience frame drops in case of unexpected memory demands.

4.4 Controls

Once we have data to look at placed in the scene, it is an excellent time to start talking about how to inspect them. No matter how good looking graphics we generate, or how efficient we are about it, poor camera controls can bring all user productivity down. In this section are proposed camera control options inspired by other Earth viewing applications.

4.4.1 Pan

The most basic mechanic is panning around the surface of the Earth. We expect that when we drag cursor along Earth surface, the camera will move in opposite direction keeping mouse cursor above the same spot. This movement needs to be generic enough to work in any camera orientation; given Earth is in the view.

We start by getting two mouse positions, for which we calculate camera rays defined by camera position and respective directions. Using these rays,

[2] `https://en.wikipedia.org/wiki/Cache_replacement_policies`

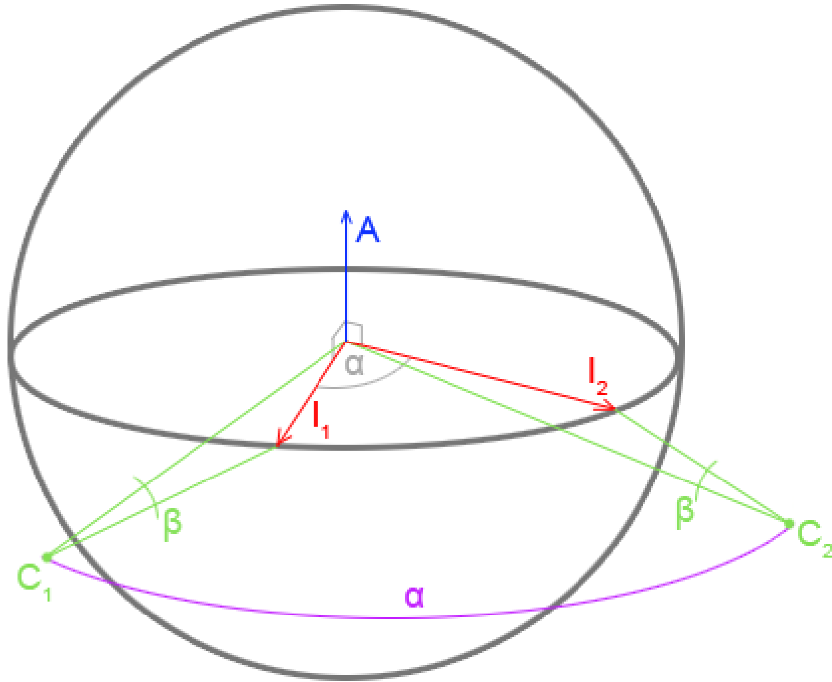


Figure 4.9: Example of camera rotation in arbitrary plane intersecting the Earth in the center. Intersection points I_1 and I_2 hold the same angle α as camera positions C_1 and C_2. Note that ray to the Earth center and to the intersection point hold the same angle β in both configurations, thus leaving the cursor at the same point on the screen.

we calculate two vectors to intersection points with Earth sphere, I_1 and I_2. At this point, we calculate axis of rotation A and angle of rotation α (figure 4.9).

$$A = I_1 \times I_2$$

$$\alpha = acos(\frac{dot(I_1, I_2)}{||I_1|| \cdot ||I_2||})$$

From these two, rotation matrix is assembled, and applied to all camera components.

The only unfortunate result of this scheme is that we have to pan along latitude lines to keep up vector aligned with the north. This is not necessarily a bad thing, as any precaution dealing with this alignment would cause problems when camera misaligns itself through different means.

4.4.2 Zoom

Now that we can pan above a specific point, we might want to zoom to take a closer look at it. Zoom function can behave a bit differently when we are moving closer or further. For starters, we would expect it to maintain cursor above the same spot on the Earth surface and move closer or further at an exponential rate. Zoom difference of 10% of the distance to intersection point seems to work well, as we do want to move fast when far and slow when close.

Misaligned with center. When camera direction vector does not point at the Earth center, we want to just move along intersection ray. In case we are zooming away, we should, at a certain distance from Earth, start aligning the camera with the Earth center to ensure normal operation point.

Aligned without intersection. Since we know the camera is pointing at the Earth center, we want to move relative to the intersection point, but also keep it at the same spot on the screen. That is not always possible because the Earth can get too small after zooming out for this relation not to have a solution.

First we calculate initial and final center distances $d_1 = ||P_1||$, and $d_2 = r + (d_1 - r) \cdot (1 + d)$, where P_1 is starting position of the camera, r is Earth radius, and d is zoomed distance as a fraction. Now we calculate angle $\gamma = \pi - asin(\frac{d_2 \cdot sin(\alpha)}{r})$, which may not exists, and that tells us there is no solution with cursor in the same spot. See figure 4.10 for context.

Once we know there is no solution, we can move along camera direction vector. It can be shown that this case happens only when zooming out.

Aligned with intersection. Continuing in the previous case as if we had found the γ angle, we calculate $\beta = \pi - \gamma - \alpha$, which is the angle we need to rotate intersection vector by to get it pointing at the new camera position. After rotating it, we rescale it into distance d_2 and set it to the camera along with new direction vector pointing to Earth center. Axis of rotation is perpendicular to vectors pointing at the intersection and starting camera position.

4.4.3 Look around

Being zoomed in on the area of interest, we might want to look around, so to say break the direct view of the Earth surface. To do that, we separate vertical difference Δ_y and horizontal difference Δ_x in mouse cursor movement and calculate horizontal angle α_h and vertical angle α_v in degrees.

$$\alpha_h = \frac{fov}{w} \cdot 2\Delta_x$$

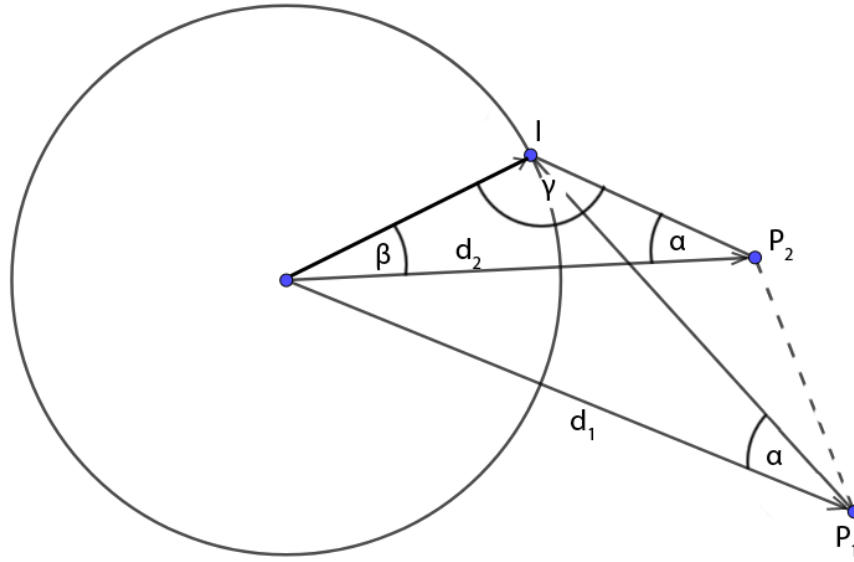


Figure 4.10: Zoom with alignment calculation.

$$\alpha_v = \frac{fov \cdot 1/a}{h} \cdot 2\Delta_y$$

where fov is camera field of view, w is screen width in pixels, h is screen height in pixels, and a is camera aspect ratio. Having these two angles, we rotate the camera around up and right vectors to get new orientation.

4.4.4 Inspection point

When we want to inspect an object on the surface, pan and turn are not exactly convenient. For this reason, inspection point rotation is added, where a user clicks on Earth surface to select a point, and subsequent mouse drag rotates the camera around that point.

Movement, in this case, is done by calculation of two angles, similarly to look around action, except here we do not have any particular scale to adhere. Vertical rotation angle $\alpha_v = \Delta_y \cdot s$, where s is constant inspection speed (horizontal analogically). Rotation axis for vertical is camera right vector, the axis for horizontal is the vector from Earth center to the inspection point.

The slightly problematic part is bounds for vertical rotation, as we want to prevent user to flip the camera upside down. We calculate vector n perpendicular to right camera and inspection point vector, facing the camera, and measure the angle between camera direction and n. Depending on the direction of the movement is clamped its amount to avoid flip or Earth collision.

```
Data: Quad tree root, Set P of previous cut
Result: Set N of new cut
Q = heap;
Q.add(root);
N.add(root);
while Q is not empty do
    tile = Q.poll();
    if N.contains(tile) then
        continue;
    end
    if tile.sufficient() then
        continue;
    end
    if tile.notReady() then
        tile.prefetch();
        continue;
    end
    tile.apply(N, Q);
end
for tile ∈ P do
    if N.contains(tile) then
        continue;
    end
    else
        tile.deactivate();
    end
end
for tile ∈ N do
    if P.contains(tile) then
        continue;
    end
    else
        tile.activate();
    end
end
```

Algorithm 2: Base slice construction

Chapter 5

Implementation

This chapter will elaborate on parts of previous chapters that were implemented, and how. Since target solution is rather large, not all made it into this work, but they stayed on the roadmap for the most part with varying priority.

We will take a look at the technology of choice, general engine organization, and then a couple of implemented features of the rendering system. Current integration of the system will be mentioned.

5.1 Technology

At the beginning of our technology selection was technically possible every engine and language. Since visio is on top of our dependency tree, we can transfer any data to it using TCP/IP, as any structures have to be redefined on receiving end either way.

Requirement for multi-platform system did cut quite a few down, but difficulties with access to low level code did final blow to most (even scene graph from JavaFX [1] and JME3 engine [2]). This essentially left us with java language using OpenGL API through JOGL [3], conveniently in same language as simulation when using near-native rendering API (OGL bindings are automatically generated).

Since JOGL usage is almost identical to its C version, all documentation and educational materials apply not only with their "how to", but also with their efficiency. In essence, C application that just renders a static scene on the screen will be as fast as JOGL application that does the same thing.

At the moment, required java version is 1.8, but it is very likely the whole project will follow any latest release (it is usually not a problem to move up in a version, as Oracle maintains backward compatibility). Required OpenGL

[1] https://en.wikipedia.org/wiki/JavaFX
[2] http://jmonkeyengine.org/
[3] https://jogamp.org/jogl/www/

version will likely be at least 4.3, but its increase is less likely, as OpenGL version support is, unlike with Java, hardware dependent.

As far as the library used for GUI, we will side with JavaFX. Even though its 3D scene graph does not suit our needs, its UI elements are much more like elements from web environment (positioning and CSS styling), which is much, much more appealing than the old way of swing [4] and awt [5].

5.2 Engine organization

New visio system is organized into three more or less separated sections.

Abstraction Side of the system that will be generally encountered by the average user will be called *abstraction*. In this space is defined a scene with no direct connection to the technology used to render elements on the screen. It gathers information from the user and sends it over to be rendered at specific synchronization time.

Implementation Technology bound side will be called *implementation*, as all rendering specifics are implemented here. This side essentially takes gathered scene information and is supposed to render it as efficiently as possible.

Renderer The significant distinction between implementation and renderer side, is that implementation section has information on all objects currently inserted in the scene, and knows how to render them, whereas renderer cares only about those that should be rendered, and triggers the rendering process.

One of the main reasons for division between abstraction and implementation sides is the asynchronous nature of the simulation. This way we can perform any changes to abstraction side scene, and only once these changes are final, we commit them to implementation side. This process is done through change events, where each event carries an instance of a mutated object, a code used to mutate it as a lambda, and data to be passed to mutating code. These are stored chronologically in a queue along with *moment of stability* marks. Now rendering system goes through this queue at the beginning of each frame and applies all committed changes to implementation scene. Neither side waits for the other during this process.

When the user performs a commit on their scene (must be a conscious decision), the whole scene graph is crawled for transformation changes, and these are added to the queue. We do this in particular because we do not want to maintain scene graph on both sides, so implementation has only terminals with final transformations.

[4] `https://en.wikipedia.org/wiki/Swing_(Java)`
[5] `https://en.wikipedia.org/wiki/Abstract_Window_Toolkit`

It is important to note, that even though objects on implementation side are created along with user commit, data for these objects may be loaded right before their usage, or never if we never use them. That means that all Earth tiles have their mesh objects created and ready right away, they are but empty shells with manual how to get their data if need be.

This leads us to the next part. We already have a list of objects that might be added to the scene before being committed, then a list of objects that could be in the scene before getting their data, and lastly, we have objects that *are* in the scene. These are inserted inside of a renderer, which keeps track of rendered scene, its properties, and does its best to render it as fast as possible.

Usually, objects that are directly inserted in the scene root are also passed in the renderer without further ado, but there are objects, such as Earth tiles, that user inserted in LOD group and those might be rendered only in specific camera position. At the beginning of every frame is performed a check with these dynamic groups using the camera for that particular frame, which might result in a change of renderer scene contents. Tiles of Earth quadtree are handled using such a structure, and Earth surface building algorithm is implemented on top of its hierarchy.

We already mentioned a couple of things that happen during a single frame, so it might as well be the perfect time to list the rest of notable remainder:

Inputs First are evaluated all input events from the platform. This includes keyboard and mouse, but also window and operating system cues. They are passed on to all registered listeners from abstraction side. These might be implemented asynchronously to render loop, but since user interaction with objects in the scene is directly tied to GL context, they had to be synchronized.

Prepare Next a preparation of the scene for this frame is called. This consists of camera fixation (camera state at this exact moment will be used for the entirety of the frame rendering), and its passing along the implementation scene. Mainly LOD groups react to it by change of their contents, but also lights that need to be uploaded to the graphics card in camera space (necessary for double precision calculations).

Update This call is performed on all objects in the renderer, which we now know to be in the scene. It usually includes load or update of any data on the GPU, but that depends from object to object. To avoid unnecessary updates, objects form a hierarchy of subscribers, where a component will notify its users in case it was changed, so they all can integrate this change when their update time comes. Because various objects share components, this update is performed on each modified component only once per frame.

Render At this moment scene is up to date and loaded on the GPU, so renderer performs all necessary draw calls. This, of course, includes numerous state changes in the form of binding updates and uniform uploads. We may have some amount of time left in the current frame,

and we might want to use this time to perform work on one of the following tasks.

Cleanup As scene changes, resources that are later no longer necessary stay allocated, taking up space for those that are vital in the current view. To make space, we can either destroy (preferably) objects that are no longer in abstraction scene, or release data of objects that are not in current renderer scene (this release can only be done over objects that have renewable data, such as static textures or meshes). We check frame time after each processed component to make sure budget is not exceeded.

Prefetch Since we never rely on resource planning (concurrently running application may exhaust all the bandwidth), we prefetch data that we might need in the following frame. This call does the only transition of data to GPU and construction of OGL objects. Prefetch for objects is scheduled during scene construction of the current frame, where every dissatisfied tile generates prefetch order. Prefetch queue is cleared at the beginning of every frame, so only fresh demands are being processed.

Each render loop described above must be executed in a synchronized manner relative to other running visio windows in same JVM. For that purpose is there render scheduler that triggers render of each window in its preferred frequency (or as close as possible), and most importantly, avoids expensive context switching.

5.3 Render passes

Scene rendering is currently done in multiple passes based on their coordinate space, used buffers, or purpose. There are three main passes based on coordinate space:

World This pass covers standard scene objects that require no particular alignment or other treatment. Transformations done in this pass are calculated directly from the main camera.

Aligned When we want to render text (or any other object for that matter) that is in world coordinate system but always aligned to face the camera with constant size on the screen, we will render it in the aligned pass. Objects in this pass have one special additional offset that is applied in world coordinates, and then is its transformation matrix treated as the transformation in this aligned 2D space.

Screen This pass is for objects that are in orthogonal screen space and aligned with one of screen anchor points (combination of vertical and horizontal lines: bottom, top, center, left, right).

Whole render pass is a little bit more complicated than changing three transformation basis, as item buffer (rendering of objects with solid color corresponding to unique identifiers for click detection) is added to the mix. It

should be noted that item buffer and main buffer share depth/stencil buffer. It proceeds roughly as follows:

1. Buffer resize to correspond with latest window size changes.
2. Main buffer clear.
3. Render world pass into main buffer.
4. Render aligned pass into main buffer.
5. Item buffer clear.
6. Render world pass into item buffer.
7. Render aligned pass into item buffer.
8. Clear common depth buffer, we are completely switching coordinate space and depth values are thus incompatible.
9. Set stencil buffer for writing.
10. Render screen pass into the main buffer.
11. Set stencil buffer for reading.
12. Clear stencil bits in item buffer. This clear is removing recorded IDs from item buffer whenever they are occluded by objects in screen space. Since not all objects are rendered in item buffer, this occlusion might not otherwise occur.
13. Turn off stencil buffer.
14. Render screen pass into item buffer.
15. Blit rendered image into screen buffer for display.

Render passes themselves differ only in the mode in which used transformation provider operates when they are being performed.

5.4 Dynamic shaders

OpenGL of version 2.0 and higher moved a bit away from fixed rendering pipeline and added support of rendering shaders that allow customized data processing in each stage of the pipeline. In new implementation are supported all types of shaders with dynamic construction on top.

When a specific feature of a shader is turned off, we would, in normal circumstances, have to pass a flag to the shader telling it not to process the data in a certain way, and this flag would be checked with every vertex or fragment being processed, introducing significant performance loss. For instance, in case we do not want to render a mesh with lighting, we have to add a flag in fragment shader telling it to skip a chunk of code dealing with lighting and using material and texture settings in a different way.

An alternative solution would be to create a shader for different uses to avoid this kind of slowdown, but with increasing amount of features a shader can support rises the number of all possible subsets and with those number of separate files to maintain. Having a switch to per vertex coloring, textures, and lighting will already generate $2^3 = 8$ different shaders, and a number of controlled features can get much higher than that.

The best solution for this issue is to create a system that will generate different code based on usage without intervention of the programmer. Luckily, GLSL [6] provides preprocessor commands that can be used to do just that. In the new system is each shader written to encompass all possible features and enclosed in conditional compilation block so it can be controlled when the shader is being compiled.

The shader is then built from a configuration that contains all compilation flags adding them as defines at the beginning of the shader source code.

As a little helper for shader writing and usage, not only feature flags can be included in shader configuration, but also layout locations of inputs, outputs, texture bindings, and various constants that could be possibly changed for different objects.

To aid the programmer with debugging of shaders, binary of Tiny C Compiler [7] was added to the system to do the preprocessor work into a debug file where it can be viewed. The programmer has then available original code with all features and branches, generated file as seen by GLSL preprocessor, and generated file after preprocessor work. The second file is necessary, as all shader compilation reporting is done with line number references relative to file state before preprocessor pass and not having to count and add added defines is a great help. The third file helps to debug all preprocessor branching seeing what actually goes into GLSL compiler.

Since a shader is completely determined by its configuration instance, we can use these instances to group all objects that would use the same shader and render them without shader switching. This grouping is one of the background optimizations that conserve resources without user intervention.

It is important to note that using too many different shaders that are overspecialized can actually be more harmful than few that do some extra work, as too much of shader switching can generate substantial strain on the system, not to mention longer loading time caused by compilation of each such shader.

5.5 Render batching

When rendering a large number of objects, the total number of state changes to OpenGL becomes a performance issue. To reduce it, we want to group together similar objects and set their common properties only once. We can imagine rendering process as a sequence of pyramids, where each floor on the left side is the assignment of property, right side is the removal of this property, and the top is a draw call.

Now we face optimization problem, where we are given a graph of these

[6] `https://en.wikipedia.org/wiki/OpenGL_Shading_Language`
[7] `http://bellard.org/tcc/`

pyramids, and our goal is to reorder them on the timeline to reduce the length of the graph to a minimum. The more advanced approach allows not only swapping different pyramids on the timeline, but also swapping floors to have the most common properties on bottom based on what objects are being batched together.

For instance, we can have two objects that share shader and lights. In case we have fixed sequence of state changes, one that does not fit currently processed object, we might happen to bundle by shader, find out that the two objects do not share material, and that results in one unnecessary lights bind. If we were to swap floors in our pyramid, we could safely bind shader, lights, and then optimally diverge on the material.

Looking at the simpler version (see figure 5.1) with a fixed sequence, we can just sort our pyramids at each level, achieving an optimal solution in $O(N \cdot K)$, where N is the number of objects, and K is sequence length. That is a fairly favorable solution, but we can still make it faster by keeping the solution from the previous frame, and apply only changes at the cost $O(K)$ per changed item, where $O(N \cdot K)$ is only the worst case.

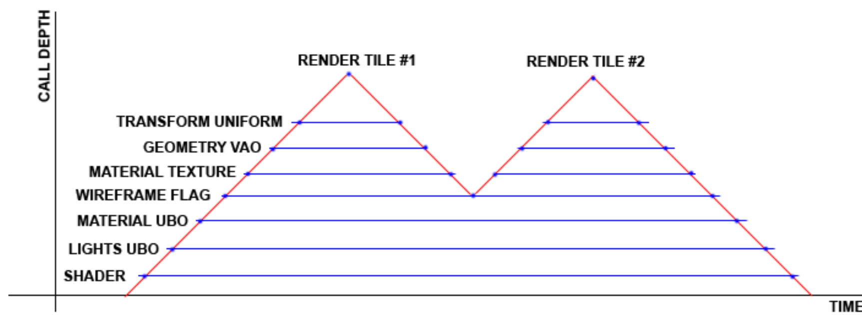


Figure 5.1: Render batching with fixed sequence for mesh objects.

The more complicated version, although with more potential for speed improvement, could be defined incrementally. We need to insert N objects into a structure, and we have $K!$ possible sequences for each object. This means optimal solution lies in a tree that has depth $h = N$ and branching factor $k = K!$. The number of possible arrangements is at most number of leaves of this tree:

$$\frac{k^{h+1} - 1}{k - 1} - \frac{k^h - 1}{k - 1} = \frac{(k-1) \cdot k^h - 2}{k - 1} \approx k^h = (K!)^N$$

We can, of course, apply various combinatoric optimization techniques (branch and bound) to reduce this number significantly, but whether it would not take more time to render using suboptimal solution or to find the optimal one is up for debate.

Because optimal solution had such questionable expected results from the start, only the simpler version was implemented. Each object type (text,

mesh, line, point) has its own hand-picked sequence, and from that is built a render batch tree where leaves are transformation nodes unique to each object. Each level has a lambda for comparing a property, lambda for getting hash of compared property (finding correct subtree in constant time), and lambda for applying that property.

When object changes, its special flag for batch related changes is set, and during render batch update phase is such object removed and reinserted. For instance, VBO contents change will not trigger batch position update, as a list of objects using that VBO did not change, so correct order of binding is not violated.

This approach is significantly better than the previous solution, which was grouping together only identical terminals. That resulted in Earth tiles doing unnecessary full rebind.

5.6 Integration

This issue was touched only lightly, as a primary focus of this work was a creation of one specific scene and underlying system, but we should still take a look at how simulation should be integrated into new visio.

User programming will be handled in a similar fashion as in current visio, using layer providers. Each layer provider has access to its own scene root, camera object to be manipulated, and user inputs for listeners and bindings. A user can then populate the scene with objects in an asynchronous manner, and commit them to be rendered on the screen.

This is of course not enough, and layer provider will need access to simulation data. Access should be managed from a single provider that handles data subscriptions and offers, so they are transferred only if someone needs them. We also need access to communication bus that allows interaction with UI elements, but a similar system is already in use, and will likely be reused for visio.

These layer providers will be dynamically loaded and initialized based on configuration XML file, but there is a possibility of loading them up at any time. The current implementation is, unfortunately, using only a couple of statically created providers with no option for outside intervention.

5.7 Scenarios

For purposes of this work, two main layer providers were created, one containing dynamic surface Earth, and one with air traffic controller view.

5.7.1 Earth layer provider

This provider contains only the algorithm for Earth surface scene building, and camera controls as described in section 4.4.

The construction algorithm works in two phases, where first is run synchronously with layer provider initialization inserting in the scene first two levels of the quadtree, and second running in the background adding all the remaining levels we have data for.

Addition of a tile consists of data wrapper construction (positions, indices, normals, texture, . . .), material preparation, and connection of all these into a tile mesh. Then LOD group is created containing this mesh, having links to previously created neighboring groups on lower levels (stitch and adjacency links). This new group is then inserted in the scene followed by a commit. Since we are constructing floor by floor from lowest levels, all groups we might want to connect to are always already prepared.

Originally, the whole scene was built and externalized into a file, but loading times were in the order of seconds, which was rather unacceptable. This way, there is something on the screen immediately, and the addition of new levels is generally faster than their loading, so no real delay was introduced. Since each layer provider has its own change queue, we do not mind large number of background threads in various other providers, at least as long as they are synchronized on the inside.

Thanks to this dynamic building process, the initialization process of Earth layer provider can be measured in the order of tens of milliseconds.

Since LOD groups are not part of the aggregated scene graph, they also appear on the implementation side, where the presented base algorithm is building the rendered scene out of them. The advanced algorithm was not added, as the generation of data for it to make any difference takes way too long, and dataset (Sentinel 2) that would be used for it became available only recently.

5.7.2 ATC view

Layer provider for air traffic controller view was added just as a validation of the engine implementation, whether it is able to handle a larger number of moving and changing labels.

To create a view similar to ATC view we have in the current system, air traffic records were extracted from currently used reference records. Each aircraft has its own record of positions, graphics group with a trail of previous positions, and data block label (static flight ID and dynamic index in the replayed record).

A background thread will check 60 times a second what aircraft should be visible and with what settings, and commits them on the screen, all based

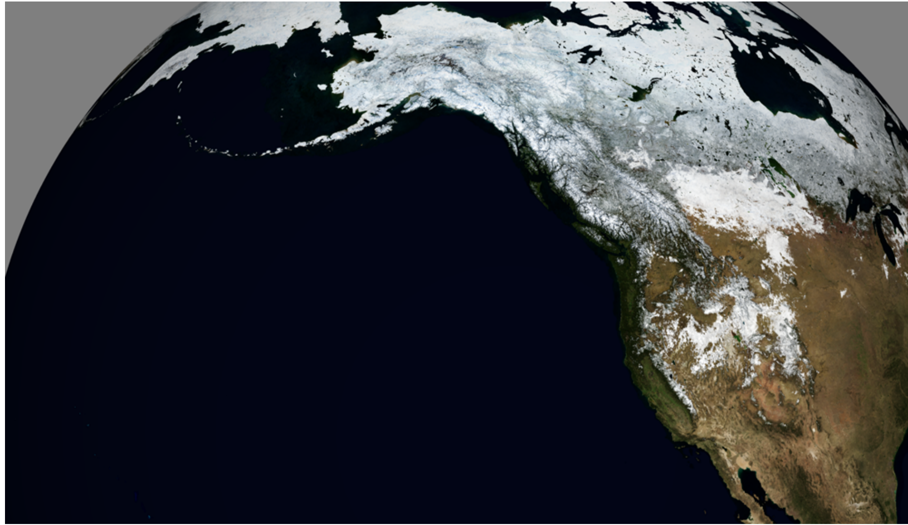


Figure 5.2: Earth surface with no exaggeration.

on artificial simulation speed of 720 (simulation time seconds per real-time second). This is slightly above the current maximum of AgentFly simulation speed.

To add some amount of context to the view, real sector boundaries are rendered as lines and fix points [8] along with them accompanied by their names. The number of fix points is rather large, so their labels can be turned on or off for visual clarity reasons.

Camera used in this scenario is a simple orthogonal view that has a "zoom" feature in form of projection plane size change, and pan.

Example of global view is in figure 5.6 with fix positions visualized only as circles with points inside, while closeup can be seen in figure 5.7 where fix positions have added labels.

[8] Named latlong coordinates used for faster and error-prone communication between ATCs and pilots. Pilots usually construct their flight plan as a sequence of these points and altitudes.

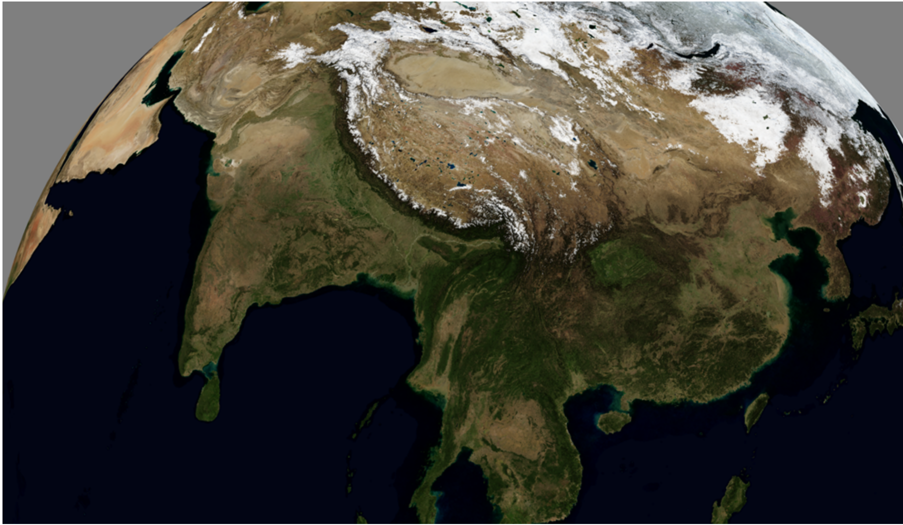


Figure 5.3: Earth surface with no exaggeration with lights turned off.

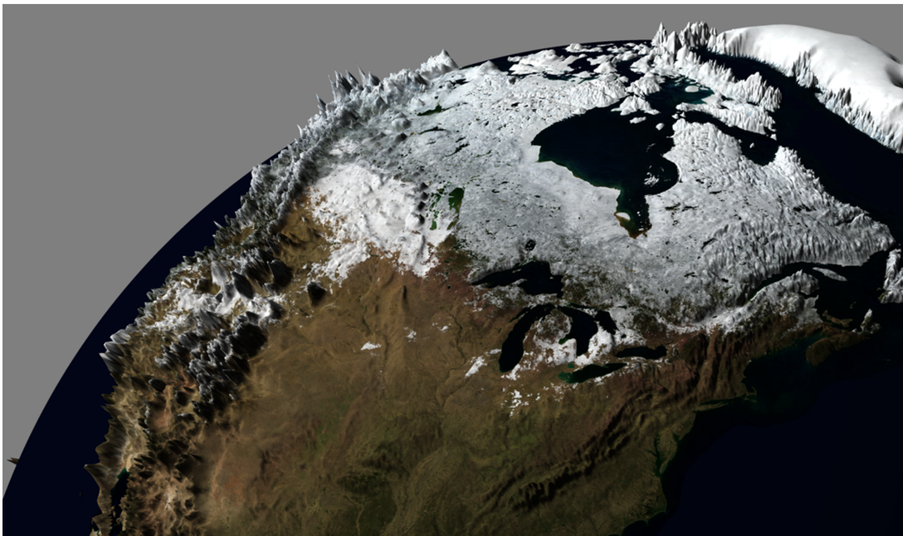


Figure 5.4: Earth surface with 100 $\times$ exaggeration, global view.

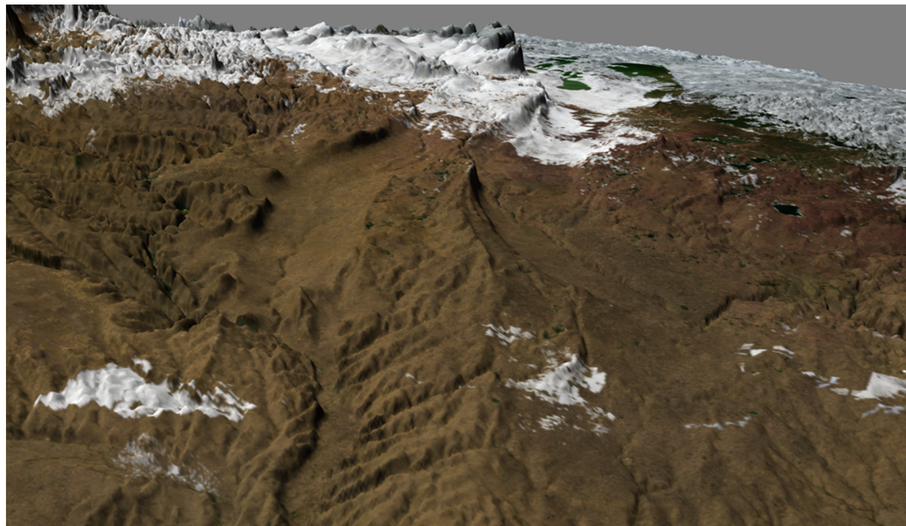


Figure 5.5: Earth surface with 100 × exaggeration, depth 6 view.

Figure 5.6: Air traffic controller view with whole Czech Republic in frame.

Figure 5.7: Air traffic controller view closeup with fix position labels.

Chapter 6

Results

This chapter presents the evaluation and measurements of the implemented system. We will take a look at missing features and known issues, and then will be discussed the performance of the generator and Earth viewing.

6.1 Missing features

In this section we list some of the features that are on the implementation road-map, but did not make it into this work. Of course, listing things that are *not* in the system could take a while, so only the most important are mentioned.

Transparency. As mentioned earlier, sectors and flight plans are displayed using semi-transparent volumes. To ensure correct color blending on semi-transparent objects, we have to render them in back to front order. That means all fragments that fall under same screen pixel need to be sorted. There is a number of possible methods dealing with this problem, many described by Maule et al. [12] (object sorting, primitive sorting, and fragment sorting).

First and simplest option is to sort objects by their centroids, which definitely helps, but the rendering of overlaps within same objects is not stable nor guaranteed correct. Objects that are intertwined may also yield wrong results. This approach becomes especially problematic with large objects, such as flight plans.

A more complicated option is to sort directly triangles. There is still danger we will get a set of triangles that cannot be sorted (we still may use triangle centroids for simplicity), but it is less prominent. This approach is quite expensive, as buffers for rendered objects may change in every frame.

The most accurate option would be so-called depth peeling [1] [2], where we render scene multiple times into different buffers. Each pass uses depth buffer

[1] https://en.wikipedia.org/wiki/Depth_peeling

[2] www.eng.utah.edu/~cs5610/handouts/order_independent_transparency.pdf

from the previous pass for testing (less fails the test), and new buffer for writing. This way we peel the scene front to back each time getting one layer. When we reach a pass that did not write any pixels, we can start composing all buffers into single image back to front. This method ensures pixel-perfect results, but in case of complex scenes requires a large number of passes.

In the end, centroid sorting will be probably the first implemented, followed by depth peeling with early exit (only fixed number of passes is made, with the last one containing all remaining geometry rendered in centroid order).

Dynamic near-far. Since our scenarios may include large-scale views as well as close up ones in a single scene, we need to set near and far attributes of the used camera depending on the current relation of camera and scene. To maximize the precision of depth buffer, these two values need to be as close to each other as possible, shifting near away when we are not close to any objects, and moving far closer when all furthest objects are behind the horizon.

To do this precisely, we need to find closest and furthest object in view frustum in each frame. That requires us to have a bounding volume over each object that appears in the scene with up to date transformation. The new system does not have this implemented, so near and far settings are determined by the distance of the camera from the surface of the Earth. This approximate approach works for Earth view scenarios, and dynamic near-far is not necessary for the others, but it still should be implemented for a general scene.

Instanced rendering. There is a large number of aircraft in Earth traffic replays, but they use only a limited amount of models. Render batching removes a lot of unnecessary hassle with these, but it can be a lot faster with use of instanced rendering. This essentially means that instead of standard draw call over VAO is performed instanced draw call, and instead of uploading a single set of matrices in uniforms, we upload arrays of them. Single draw call can then take care of rendering of hundreds of objects, significantly increasing the throughput of the system.

Test traffic replays showed that the strain from traffic rendering is not negligible, and instanced rendering would alleviate a lot of it, especially on the weaker hardware used for presentations outside of the office.

Normal maps. Since we have elevation data in tiles that have the exact same dimensions as diffuse tile textures, we might convert these elevations in normal maps and increase visual fidelity of tiles at very low cost. This would help especially with hill shadows that pop in with higher level tiles being used, as these would be visible on lower levels with no extra geometry.

Implementation of normal maps specifically for Earth tiles is a little bit more complicated, as used normals would not be correct on stitches. Because

of issues like this, normal map addition in the system got low priority.

Geometry aggregation. Visualized data can be quite often separated into small pieces of geometry, aircraft trail line for example, and they produce unnecessary strain on the system. We can take all these little lines of same color and width, and merge them into a single object. Even though we will be forced to do buffer update for this object in nearly every frame, it is still faster than performing hundreds of draw calls for separated lines.

6.2 Known issues

In this section will be described some of the unresolved issues with the new system, and ideas how to go about solving them.

Garbage collector. Because the language used is Java, one with an automatic memory management and garbage collection, we encounter issues with pauses dedicated to the cleanup of dead objects. Issues with standard garbage collectors are that they usually have one or more stop-the-world phases causing frame delays. Luckily for us, Java 1.7 introduced new concurrent garbage collector intended for real-time applications.

G1 GC is running in parallel with the application and is cleaning up objects without frequent pauses. Unfortunately, it is not always possible to avoid pauses, and they have to be performed at least occasionally. This manifests in rendering lag that is out of our control.

We can set G1 GC to try to limit its pauses below a specified amount of milliseconds, but that serves only as a hint, and it is usually not followed. We can also try to force garbage collection at the end of each frame, hoping it will do its work in between waiting for GPU synchronization barrier, but system call for GC triggers long pause that flat out butchers our rendering performance.

We are basically out of options on JVM side of things and have to reduce the amount of generated garbage. In early system tests, GC pauses were above one second every five seconds of runtime, which was cut down drastically to about 100ms in intervals that depend on current activity, but that is still quite uncomfortable.

Profiling done over current system implementation showed that a large number of collected objects is generated by in-built Java collections (HashMap, LinkedList, TreeSet, etc.), that use objects for its inner structures. This means we have to either modify these collections to use object pooling [3], or write new collections that do not use any additional objects, and work only with arrays of primitive types and stored objects.

[3] https://en.wikipedia.org/wiki/Object_pool_pattern

Modification of inbuilt collections is not made exactly easy, but it should not be impossible either, as methods that create or stop using inner helper objects are often exposed for an override. Such a modification can come at a performance cost, since specifically in Java, the new operator takes only a couple of cycles, while pool object retrieval can consume hundreds, and it gets especially nasty in case of a concurrent environment.

Implementation of these hotspot collections in garbage-less versions can be quite time-consuming, mostly because we are trying to match the performance of well thought out competition that evolved over many years.

The last option is to look for salvation in a library such as Trove [4], which may or may not help, depending on specific replacements. Some of the Trove collections are garbage-less and could benefit our cause.

Insufficient GPU memory. When moving close to Earth poles in the global scenario, a large number of tiles will be forced on screen (even though their contribution is questionable, to say the least). This results not only in a large number of rendered tiles but also in large quantities of tiles forced in GPU memory due to maximum +1/-1 adjacency. The total number of tiles is over 500 when viewing poles at depth six, resulting in GPU memory swap into main memory and drastic framerate drops.

In the ideal case, we would replace poles with caps that subdivide more efficiently, but that is difficult to do in the generic quadtree, and we would reintroduce all mapping issues mentioned in data representation chapter. What seems like a more viable option is to generate maximum adjacency depth difference connections in a way that closer to poles means we can afford higher resolution difference. This approach along with lower maximum resolution at poles would reduce the number of ready tiles down to 80 (assuming pole depth 5 is sufficient and we allow depth difference of 2), which is much more manageable.

6.3 Generator performance

Since the Earth surface data generation is an integral part of global scenarios, and it turned out to take a non-negligible amount of time, we should take a better look at overall performance of used generator.

Generation was implemented as described in section 3.4. Raw data for mesh representation is constructed for each tile separately (texture coordinates, indices, positions, normals, bounding volume, occlusion volume), and stored in resource module under a unique key.

The whole process can take up to 24 hours for mere quadtree depth 6. The main reason for this slow generation is elevation map queries. If we were to

[4] `https://bitbucket.org/trove4j/trove`

perform the same process over same mesh resolution, we would need only minutes to produce a quadtree of equal depth.

There sure is room for improvements in current elevation map implementation, but it is a good idea to try to beat the problem over the head with raw hardware power before diving into optimizations that may or may not help. The main reasoning is that we will generate this dataset only once (assuming change of raw data will be rare), and thus it is not a problem if it takes a couple of weeks to finish. However, it is a problem to spend a couple of weeks perfecting generation algorithm to save percents.

First, we take a look at multi-thread parallelization. Because elevation map implementation is thread-safe, we can run construction on all tiles in any order, as there are no other dependencies between them. Storage is not an issue either, because we are storing into a binary file under a key, and it is not relevant where in it is stored what particular tile component. Of course, we should proceed in more or less ordered fashion simply for elevation query caching purposes, but it is not necessary.

We can see in figure 6.1 that threads working together do not indeed influence each other all that much. Overall time it takes to generate the quadtree in the maximum depth of 6 is split in the number of physical cores assigned to the generator, while virtual threads do not help all that much anymore.

This approach cuts down generation of depth 6 to bearable 5 hours, but what if we were to construct quad tree for depth of 12, matching our current satellite dataset. To get rough estimate, we can start by calculating time for single tile $t = \frac{time}{tile\ count}$. Time is our measured 19 332 seconds, and number of nodes in complete quad tree is $n = \frac{4^{h+1}-1}{3} = \frac{4^8-1}{3} = 21\ 845$ — our tree is half empty, so total number of nodes to consider is 10 922 — leaving us with $t = \frac{19\ 332}{10\ 922} \approx 1.77$ second per tile, and $t \approx 7.08$ seconds for single threaded work.

Now we need to calculate the number of tiles being generated in a complete tree, but we should keep in mind, that we will not generate tiles for sea. For this estimate, we can take a look at the number of tiles at the deepest level, and start dividing them by four to get to lower levels. Therefore we are solving sum of geometric series $S_n = a_0 * \frac{1-r^n}{1-r}$, where $a_0 = 11\ 626\ 643$ is number of tiles in depth 12, $r = 1/4$ subdivision factor, and $n = 12 - 6$ depth difference we are accounting for. The total number of tiles is then $S_n \approx 1.54984059521484375 \cdot 10^7$, and time to generate their data about 717 days of single-threaded computing time.

Two years is a somewhat unacceptable amount of time, but because we are generating the data independently into a keyed binary file, and because we can chain multiple of these files together in the new system, we can run the generation process on any number of computers without any additional work.

In means of AgentFly, if we employ just five of our six core machines, total

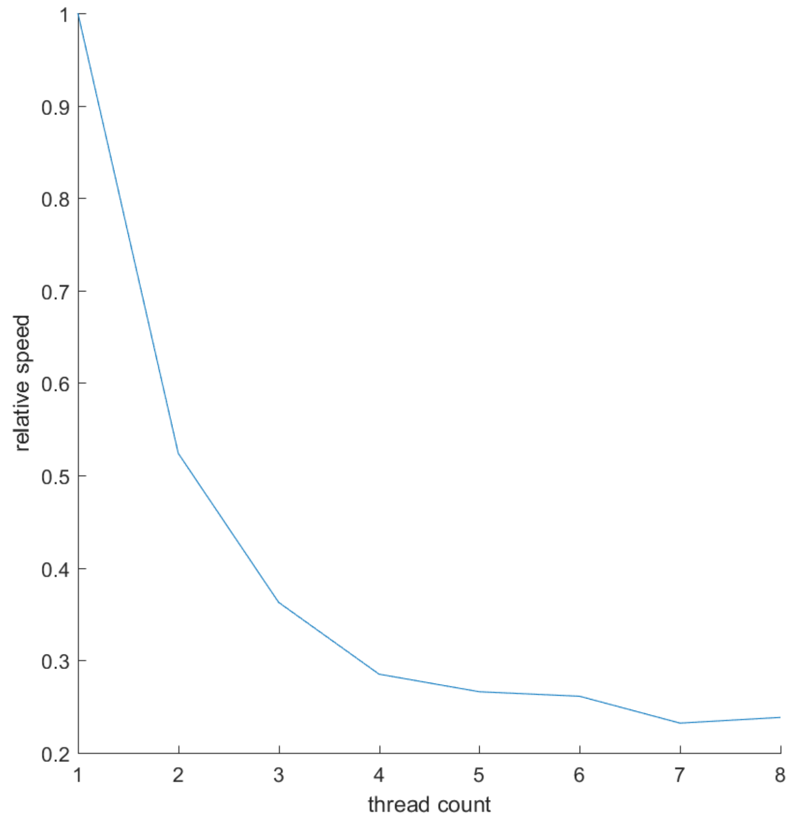


Figure 6.1: Time to generate depth 6 quad tree Earth surface, values relativized to single thread performance. Test machine has 4 cores and 8 threads.

tile gets cut down to $\frac{717.519}{5 \cdot 6} = 23$ days. If we run the generator only outside of work hours, it will take about a month, which is already manageable.

It is important to note that textures are keyed separately, so only single computer needs to have access to Sentinel 2 dataset, and remaining ones need only elevation map (≈175GB).

6.4 Earth viewing performance

At last, we will investigate the performance of Earth surface viewing in the new system.

Scene. Measurements were performed on complete Blue Marble dataset scene with elevation map of the same depth and resolution 128×128 per the corresponding tile, exaggeration set to $100\times$. Decimation vertex maximum was set to 10 000 (about 18 000 triangles) per tile. Each tile in the scene has

set identical ambient and directional light (sun).

As far as renderer properties go, all frames were rendered in 8K resolution with linear down-sampling. All textures have generated mipmaps and anisotropic filtering 16×. Used lighting model was complete Phong [5]. Target framerate was set to 60 FPS with preferred GC pauses on $10ms$, *vsync* off. Safety scene rendering budget was set to $10ms$ to account for unknown expected last operation duration (setting it higher results in more frequent frame drops under 60 FPS).

Camera. To ensure more or less consistent results, single 30 seconds camera movement was recorded and replayed on abstraction side. Samples will not be exactly $1:1$, but at the scale of 2000 frames, the difference should not be visible even if we consider GC interruption. Camera replay was started right after Earth layer provider initialization.

Fidelity. As mentioned in subsection 4.3.2, constant quality factor was added to desirability heuristic.

$$dh = \frac{l^2/(\frac{h}{t} \cdot Q)}{d^2}$$

$Q = 1$ does not change the heuristic at all, $Q < 1$ will decrease final fidelity, $Q > 1$ will increase final fidelity.

Test computer. Measurements were performed only on a single machine (see table 6.1), as we are interested in relative values rather than absolute performance. We can always adjust quality factor to fit the used hardware and achieve similar results. This was tested on multiple computers, but their results did not differ enough to justify their presence in this thesis.

OS	MS Windows 10 Enterprise x64	CPU	Intel Core i7-7700 CPU @ 4.2GHz
Cores (threads)	4(8)	RAM	64GB
GPU	GeForce 1070 8GB	Driver version	388.13
Java	1.8.152	Graphics clock	1506MHz
Bandwidth	256.3 GB/s	Shader units	1920
Pixel fill	96.4 GPixel/s	Texel fill	180.7 GTexel/s

Table 6.1: Test computer

6.4.1 Fixed quality

First, we will take a look at the scenario with $Q = 1$. Please note that this scenario has different camera trajectory than one used for comparative charts. We start with the camera at three times Earth radius away from Earth center, zoom in on maximum depth and wait until it is loaded (frame 400). Then we

[5] `https://en.wikipedia.org/wiki/Phong_reflection_model`

start panning around, and after a couple of seconds look around the horizon from a fixed position (frame 1600).

In figure 6.2 are full frame times in particular frames. This number includes the whole process described in section 5.2, not only rendering. We can see that most frames that take longer than target $16.66ms$ are those with garbage collector pauses. These stutters are unfortunately noticeable, and we should try to reduce them as much as possible (see section 6.2 for an in-depth discussion on GC issue).

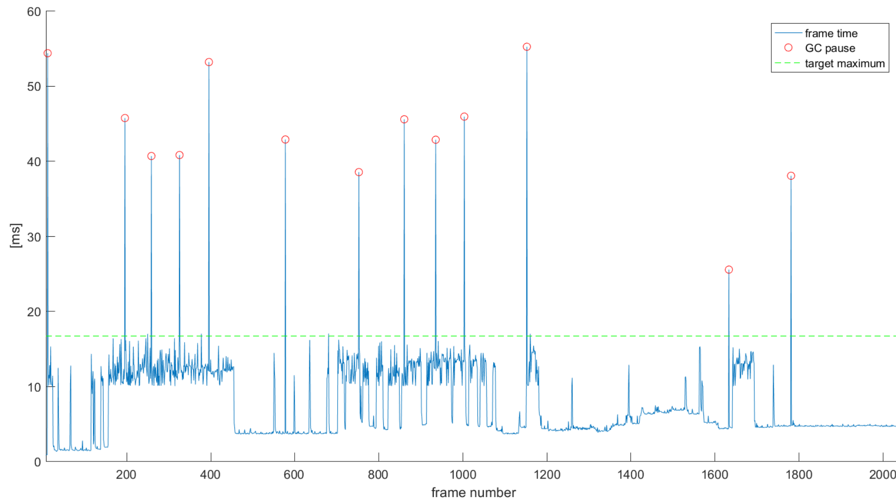


Figure 6.2: Frame time for $Q = 1$, green line shows target frame time of $16.66ms$, and red circles mark garbage collector pauses.

Dissatisfied tiles figure 6.3 shows us a number of tiles that were considered for improvement in that particular frame but were not available. We can see how this number rapidly grows as we zoom in at frame 200, reach its peak after we stop zooming in, and then decrease as prefetch process starts catching up. When we start panning at frame 600, we can see how quadtree cut gets expanded only slightly. Since panning was done with the camera facing Earth center, dissatisfied tiles did not have time to appear in the view. GPU memory occupancy shadows decrease in dissatisfied tiles count, as expected.

Looking at the particular distribution of tiles in the current cut (see figure 6.4), we can see how depths $0 - 2$ disappear almost immediately, $3 - 4$ get to a stable amount (opposite side of the globe changes less frequently), $5 - 6$ responding to the current view. At frame 400 is visible how VFC kicks in reducing requirements on tiles that are not in view, as those are always assumed to be of sufficient quality.

The same story goes for the number of rendered tiles, where those at lower depth are rendered in the beginning, but are quickly replaced by maximum depth or culled away by VFC. Between frames 400 and 650 is camera too close to one particular tile, and nothing else is rendered. Then we zoom out

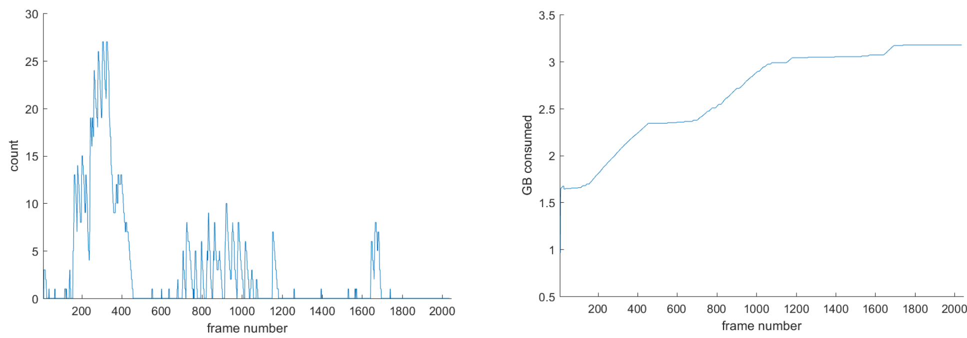


Figure 6.3: Number of dissatisfied tiles over time (left), amount of consumed GPU memory in GB (right).

a little bit having different maximum level tiles in the view. Around frame 1200 is zoomed in for a moment, and then the look around where VFC does not cover horizon tiles.

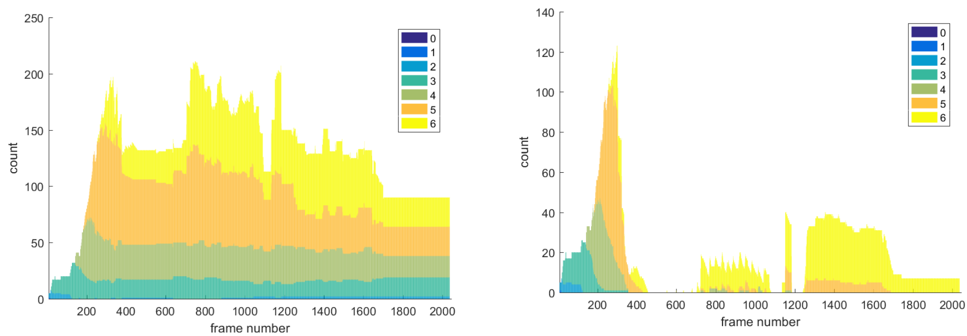


Figure 6.4: Current quad tree cut size in number of tiles per depth (left), number of rendered tiles from current cut per depth (right).

6.4.2 Comparative

We were discussing quality factor equal to one, which is base desirability heuristic. Now is the time to take a look at comparative measurements of different factors. These graphs are fairly cluttered and serve only for an overview of the general relation between quality factor and performance.

In figure 6.5 are frame times for different quality factors, and it is clear that system stays at 60FPS, with only outliers in the form of garbage collection pauses. We can also see that render times are very similar for all settings, at least when nothing is being loaded onto GPU (lowermost line).

When looking at dissatisfied tile counts in figure 6.6, it looks almost as if chart for $Q = 5$ was scaled down for the others. Cut size in figure 6.7 could at first glance suggest similar behavior, but in first 400 is quite clearly visible how lower qualities scale at a steady pace, and then higher qualities hit system limits for tile loading ($1 - 5$ do almost overlap). The same effect

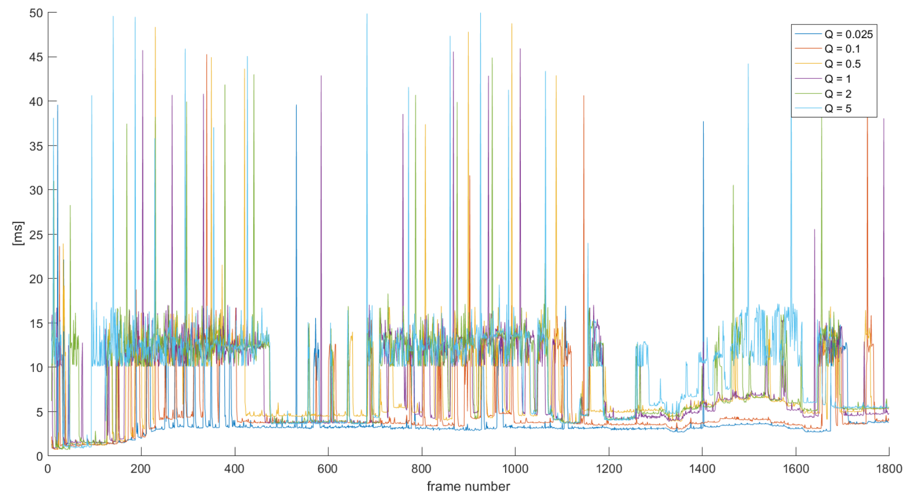


Figure 6.5: Frame time for different quality factors.

is visible in this figure as well as in the number of rendered tiles in figure 6.8.

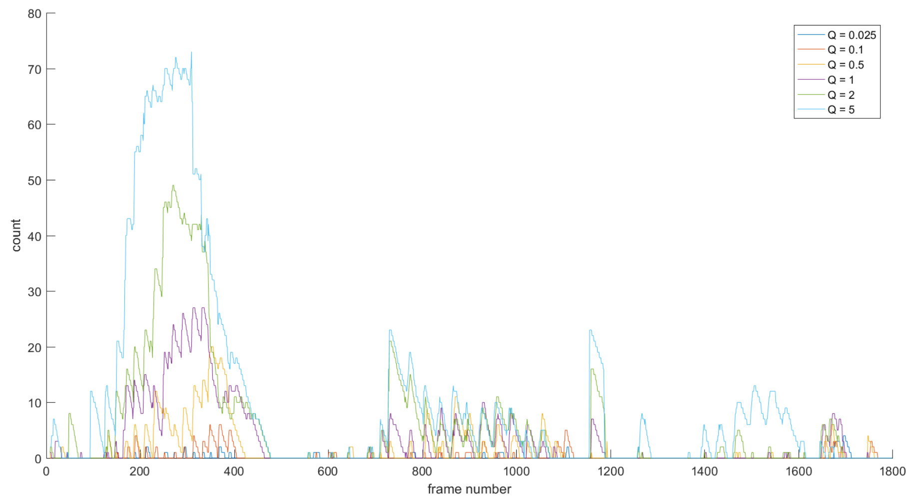


Figure 6.6: Number of dissatisfied tiles for different quality factors over time.

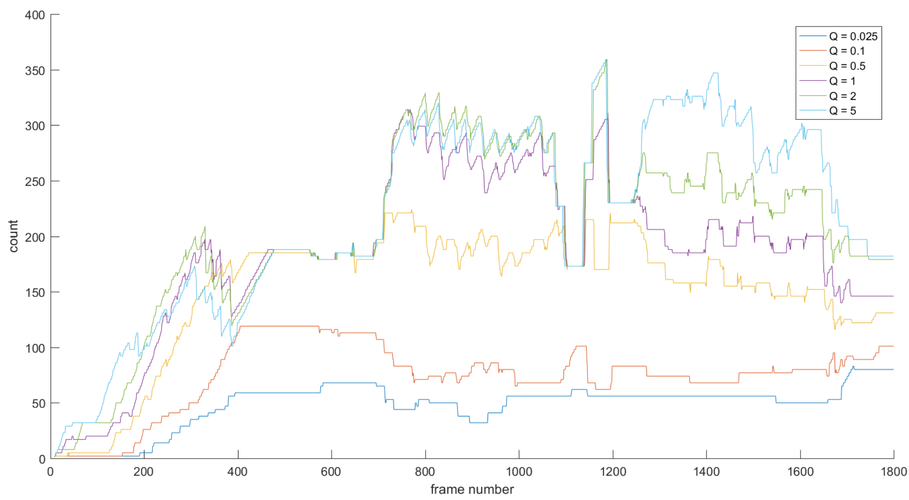


Figure 6.7: Size of the cut for different quality factors over time.

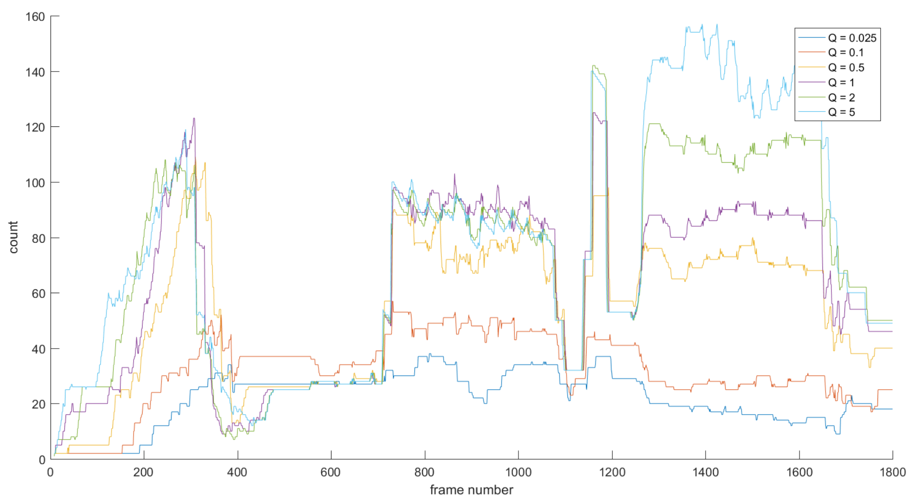


Figure 6.8: Number of rendered tiles for different quality factors over time.

Chapter 7 Conclusion

Throughout this work, the project AgentFly and circumstances of its visualization system upgrade were introduced.

At first, we took a look at all specifics of the current system, all its specifics and flaws, and what are requirements for its replacement. Elaborated requirements were not only rigid general ones, or specific functional, but also requirements set by its past and future usage. When talking about such a visualization system, we had to investigate related works that deal with large scale scenes as well as works on budget rendering topics.

When displaying planetary scene, we encounter large amounts of data, so we analyzed the best representation of such datasets. Specific collections of data were brought up and adapted to our environment, with details on how to deal with their issues. Processes for conversion of raw data to 3D scene objects were drafted.

Furthermore, rendering of large-scale scenes and specifically Earth surface was elaborated on. Spatial precision issues were taken apart, and multiple solutions were proposed. To put the only subset of generated data on the screen, an algorithm for scene slice construction was devised and broken down. Impact of concurrent tasks and hardware performance was accounted for, and ways to deal with it were introduced. Controls of the camera in such a large scene were described as far as their expected behavior goes, and then calculations necessary for their realization were presented.

Lastly, the actual implementation of previously proposed solutions was described on specific platform and technology with several optimizations being mentioned. Earth view and controller view scenarios were shown in their current state in the new system. We took a look at missing features and known issues of final implementation, suggested ways to deal with them, and finally the measured performance of the system and its parts.

The current implementation of the new system is not ready for exchange with previous one, at least not in full extent, but most of the groundwork on the engine is laid out, and Earth surface visualization is usable for the most part. Even though we cannot display all the data, it is possible to build

the most common and used scenarios while more specific features are being added.

7.1 Future plans

This section contains a short overview of plans for near future with the development of the new system. These are not necessarily in order, but they should be addressed sooner or later.

Known issues. As described in section 6.2, we need to take a proper look at garbage collection pauses, mainly at sources of temporary objects and ways to reduce their production. Even though it is only noticeable, and far from problematic when the system is being used, it should be still reduced to a minimum. Integration of Trove library is the least that can be done.

Issue with number of tiles around poles should be definitely resolved, not only because it pretty much breaks viewing experience (even though we never look there), but also because implementation of steps larger than one can enable us to add narrower data cones around hot-spots, thus having to supply lower volumes of data to costumers.

Missing features. From missing features (see section 6.1), transparency is a must have, and needs to be addressed in at least the minimal centroid sorting manner. Other mentioned points are rather nice-to-have and will be likely added over time when the necessity or occasion arises.

The main point is that we do not pay as much attention to other listed missing features, that is that they are oriented to performance improvements, and that is more or less sufficient at the moment. Of course, if we use weaker hardware, we will likely experience a certain amount of performance loss. However, it runs sufficiently on a notebook integrated graphics card, so it makes sense to start optimizing once rendered scenes get too complicated.

Scene construction extension. In section 4.2 is described base construction algorithm (which is implemented), and extended version that deals with known potential issues. This change should be added sooner than later, as it might require changes on abstraction side objects, and those might interfere with any transferred code from the old system.

Extended construction algorithm should enable deeper and more complex scene cuts on weaker machines, as it better utilizes currently available resources and does not reset state after each frame. When tuning construction, we should take a look at desirability heuristic, and possibly introduce performance feedback loop to quality factor.

Data. Because our elevation dataset is now as good as it will likely get anytime soon, our only concern is tile data generator speed and Sentinel 2 missing data. For the former, we can, as described in section 6.3, generate slowly all tile data without having specific diffuse textures, as they can be added later.

For latter, at least a proper cleanup process needs to be derived for missing sea samples with smooth transitions. That means loading and processing of coastal vector lines and implementation of color filters directly in the new system (color correction values were calculated manually in Matlab).

GUI. The new system needs to be appropriately bound with GUI environment, allowing communication of UI elements with visualization layer providers, but also with simulation scenario providers (modules that control simulation).

Since technology is already selected (JavaFX), we only need to establish UI architecture. This means well thought out connection between all three systems (visio, simulation, GUI), but also templates that allows easy addition of new control elements.

Simulation binding. Not only controls need to be communicated between visio and simulation, but also data. Large amounts of data. For this will be necessary to implement module system for offer/subscription of data. These data modules will have to handle static data (flight plans registered from pilots), dynamic generational data (currently valid flight plans), but also incremental data (history of flight plan changes).

This process requires involvement of entire AgentFly team to avoid any unpleasant surprises in the form of rare specifics and legacy systems.

Bibliography

[1] Sahr K, White D, Kimerling AJ. *Geodesic discrete global grid systems.* Cartogr Geogr Ing Sci 2003:30(2):121-34.

[2] Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R. *Planet-sized batched dynamic adaptive meshes (P-BDAM).* Proceedings of IEEE visualization, VIS'03, Seatle, WA, USA: IEEE Computer Society; 2003, p. 147-55.

[3] Snyder JP. *Map projections — a working manual.* Washington, DC, USA; US Government Printing Office; 1987.

[4] Mahdavi-Amiri A, Samavati FF, Peterson P. *Categorization and conversions for indexing methods of dicrete global grid systems.* ISPRS Int J Geo-Inf 2015;4:320-36.

[5] Mahdavi-Amiri A, Alderson T, Samavati FF. *A Survey of Digital Earth*, Computers and Graphics, vol. 53/(2015), pp. 95-117.

[6] Sellers G, Obert J, Cozzi P, Ring K, Persson E, de Vahl J, et al. *Rendering massive virtual worlds.* SIGGRAPH 2013 courses. ACM; 2013.

[7] G.H. Dutton. *A hierarchical coordinate system for geoprocessing and cartography.* Lecture notes in earth sciences, Springer, Berlin, Heidelberg (1999)

[8] Lee M, Samet H. *Traversing the triangle elements of an icosahedral spherical representation in constant time.* In: Proceedings of the 8th international symposium on spatial data handling, 1998. p. 22–33.

[9] A Mahdavi-Amiri, FF Samavati, P Peterson. *Categorization and conversions for indexing methods of discrete global grid systems.* ISPRS Int J Geo-Inf, 4 (2015), pp. 320-336

[10] Thorne C. *Using a floating origin to improve fidelity and performance of large, distributed virtual worlds.* In: Proceedings of the 2005 international conference on cyberworlds, CW05, 2005.

[11] Desai M, Ganatra A. *Survey on Gap Filling in Satellite Images and Inpainting Algorithm.* In: International Journal of Computer Theory and Engineering, Vol. 4, No. 3, June 2012

[12] Maule, Marilena, João L. D. Comba, Rafael P. Torchelsen, et al. *A Survey of Raster-Based Transparency Techniques*, Computers and Graphics (Pergamon), vol. 35/no. 6, (2011), pp. 1023-1034.

[13] Gobbetti E, Marton F, Cignoni P, Di Benedetto M, Ganovelli F. *C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering.* Computer Graphics Forum, 25: 333–342. doi:10.1111/j.1467-8659.2006.00952.x (2006)

[14] Wimmer M, Wonka P. *Rendering Time Estimation for Real-Time Rendering.* Proceedings of Eurographics Symposium on Rendering 2003, ACM SIGGRAPH, June 2003.

[15] Kooima R, Leigh J, Johnson A, Roberts D, SubbaRao M, DeFanti TA. Planetary-Scale Terrain Composition. IEEE Transactions on Visualization and Computer Graphics. 2009;15(5):719-33.

[16] Livny Y, Kogan Z, El-Sana J. *Seamless patches for GPU-based terrain rendering.* The Visual Computer. 2009;25(3):197-208.

[17] Losasso F, Hoppe H. *Geometry clipmaps: Terrain rendering using nested regular grids.* NEW YORK: ASSOC COMPUTING MACHINERY; 2004.

[18] Liu X, Rokne JG, Gavrilova ML. *A novel terrain rendering algorithm based on quasi Delaunay triangulation.* The Visual Computer. 2010;26(6):697-706.

[19] Kidner DB, Ware JM, Sparkes AJ, Jones CB. *Multiscale Terrain and Topographic Modelling with the Implicit TIN.* Transactions in GIS. 2000;4(4):361-78.

Appendix A

CD contents

In this chapter will be described contents of CD attached to this thesis.

A.1 Files

All files on the disc are compressed in *.zip* archive in the root folder. The list below contains all notable files and folders with a short description of their purpose. Please note that documentation and source files are a "snapshot" that does not include code used for argument handling in attached distribution.

/thesis.pdf
: PDF document with this text.

/javadoc/index.html
: Generated documentation entry point. Documentation was generated using Java Oracle generator [1] over attached sources.

/sources/common/
: Source codes for communication interfaces between abstraction and implementation sides. These interfaces have suffix *A* to denote their purpose.

/sources/facade/
: Source codes for abstraction side classes, word *facade* used because *abstract* is Java keyword. These classes have no suffix.

/sources/glsl/
: GLSL sources before compiler processing and added defines. Files for same shader have same name with extension defining their purpose (*.vert* vertex shader, *.frag* fragment shader, *.comp* compute shader).

/sources/GUI/
: Package intended for GUI related classes, currently contains only single frame entry point.

/sources/implementation/
: Source codes for all implementation side classes, also contains renderer

[1] `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html`

objects. Classes with a direct counterpart on abstraction side have either *Data* or *Object* suffix depending on their purpose, source data wrapper or OpenGL object wrapper, respectively.

/sources/testing/
Source codes for all test classes, implemented scenarios, and data processing code.

/sources/testing/earth/
Earth data processing classes, elevation and imagery source tile data, and mesh generation.

/sources/utils/
Source codes with utility classes used throughout the project.

/distribution/lib/tcc.exe
Tiny C Compiler [2] for shader debugging.

/distribution/src/glsl/
Shader sources loaded at runtime, may be modified to change rendered scene (normals display, camera distance, $\cdots$).

/distribution/natives/
Native libries for JOGL.

/distribution/visio_lib/gluegen-rt.jar
Generator [3] of JNI [4] code for specific platform during application startup. This dependency is used for JOGL, JOCL, and JOAL.

/distribution/visio_lib/jai_imageio-1.1.jar
Library [5] for additional image formats, currently used only for $tiff$ file processing.

/distribution/visio_lib/java-data-front.jar
Library [6] for wavefront [7] OBJ and MTL file support.

/distribution/visio_lib/jogl-all.jar
JOGL [8], Java OpenGL API bindings.

/distribution/visio_lib/vecmath.jar
Vecmath library for simple vector math, pulled out of J3D project for compatibility, to be replaced by newer version or in-house fork.

/distribution/work/
Source and temporary data files.

/distribution/visio.jar
Compiled visio distribution, requires specific set of arguments to run.

/distribution/run.bat
Batch file running sample scenario with predefined arguments for JVM and visio itself.

[2] https://bellard.org/tcc/
[3] https://jogamp.org/gluegen/www/
[4] https://en.wikipedia.org/wiki/Java_Native_Interface
[5] https://github.com/jai-imageio/jai-imageio-core
[6] https://github.com/mokiat/java-data-front
[7] https://en.wikipedia.org/wiki/Wavefront_.obj_file
[8] https://jogamp.org/jogl/www/

A.2 Distribution

This section describes details regarding attached test distribution of the system. First, we will take a look at attached data and their properties, then requirements for successful execution will be stated along with possible arguments. Controls within running application will be listed, and outputs elaborated on.

A.2.1 Data

There are three data packages included in the distribution. Two for mesh data at zero and 100 exaggeration, and one containing textures. Each of these packages is split into two files, one with $-CLF$ suffix (keys, sizes, and offsets), and one with $-DF$ suffix (raw data). Mesh packages contain vertex positions, texture coordinates, indices, normals, and bounding/occlusion volumes.

Both generated datasets are for quadtree of depth 4 with elevation map in resolution 128×128. Applied decimation had maximum vertex count set to 10 000, the base error of 100 meters, and maximum triangle area $A = 10^{11}$.

A.2.2 Execution

Execution of this distribution requires at least Java 8, graphics card with OpenGL 4.3 capabilities, and at least 2GB of available graphics and system memory. It may be necessary that administrator privileges will be necessary, as the code needs access to native libraries and generated gluegen executables.

For the execution of $visio.jar$ distribution is necessary to pass a number of arguments, some of the notable ones are listed below:

- autoplay [$boolean$], when set to true, camera replay layer provider will initiate camera replay immediately after successful initialization. This option is intended for comparative benchmarks.
- displayOBB [$boolean$], when set to true, the wireframe of used oriented bounding boxes for tiles will be displayed.
- windowCount [$0 < integer$], a specified number of windows with identical contents will be generated. It is not intended use case to have multiple Java virtual machines running in parallel; all AgentFly instances should be under a single environment.
- qualityFactor [$0 < float < Inf$], quality coefficient described with desireability heuristic. Neutral value is $Q = 1$, lower value results in lower quality, higher value in higher quality.
- resolutionMultipler [$0.5 \leq float \leq 2$], resolution of render buffer is screen resolution multiplied by this factor. Since linear interpolation on four NN samples is used, it makes no sense to set values outside of half and double resolution.

- targetFPS [$0 < float < Inf$], frames per second render manager should strive for. Maximum frame budget is statically set to $10ms$, so there will be frame drops on higher desired frame rates due to tile loading, but 60 FPS should generally be a minimum.
- exaggeration [$boolean$], when set to true, a dataset with exaggeration $e = 100$ will be used for all tile meshes.
- Xms2g, JVM argument setting starting heap space size, in this case to 2GB.
- Xmx4g, JVM argument setting maximum heap space size, in this case to 4GB.
- XX:+PrintGC, JVM argument that enables garbage collector debug output. It generally contains marks when various phases start and end.
- XX:-PrintGCTimeStamps, JVM argument that adds duration timestamps to garbage collector output.

Do not change the order of arguments in $run.bat$. The correctness of given arguments is not checked, and passing invalid values may result in a crash.

A.2.3 Controls

In this section will be listed specific control binding from implemented layer and camera providers.

Camera control. Bindings for camera movement are defined in testing package, Sphere Camera Controller class.

- Inspection mode is activated by mouse wheel press on the surface of the Earth, followed by drag. Movement up and down will tilt the camera, while a movement to the left and right will rotate it around inspection point axis. Tilt is constrained to about 90 degrees.
- Press of the right mouse button will initiate mode in which drag down and up results in smooth zoom in and out, respectively. In case the press was initiated over Earth surface, zoom function will try to keep this point in the same spot on the screen.
- Drag with left mouse button pans the camera around the globe. The camera can be in any configuration as long as the cursor is over Earth surface. In case cursor leaves it, panning will be interrupted and resumed in case cursor returns.
- Step zoom can be performed either with scroll wheel (forward is zoom in, backward is zoom out), or using double click (left mouse button zooms in, right mouse button zooms out).
- To look around camera position, hold CTRL and drag with left mouse button across the screen. Vertical movement is camera tilt, and horizontal movement is rotation around vector from the Earth center to camera position. This movement is not restricted.

Camera replay. Bindings for camera replay are defined in testing package, Camera Replay Layer Provider. Record in the file is automatically loaded during provider initialization.

- Toggle recording with $CTRL + R$. Recording always adds at the end of the currently loaded record.
- Toggle replay with $CTRL + P$. Replay starts either at the beginning of the current record or at the position it was previously paused on. When replay reaches the end, it stops, and pointer in the record is set to the beginning. If replay is resumed on out of bounds pointer, it immediately stops and is reset.
- To load record from file (*camera.dat*) press $CTRL + L$. The loaded record is always added at the end of current one.
- To store current record to a file (*camera.dat*) press $CTRL + S$.
- To delete the record currently residing in memory press $CTRL + D$. This action does not affect the record in the file.
- To return to the default/initial camera position press *home*.

Scene. Bindings for scene control are defined in the testing package, Earth Layer Provider.

- To properly see what tiles are loaded and in what level, we can switch between fill and wireframe render mode by toggling W key. This wireframe will still perform backface culling, and it will not remove Earth occlusion.
- Earth has in its scene a fixed directional light source. This light can be turned on or off by toggling L key, allowing us to see darker areas of the surface.
- Currently rendered tiles can be viewed by camera state fixation by pressing C key. When the camera is fixated, its state from the time of fixation will be used for scene construction purposes until it is released with a repeated press of C key.

A.2.4 Outputs

Running application produces three kinds of outputs: benchmark files that record the state of the dynamic Earth scene in text form, console outputs with GC and frame time, and used shader files.

Benchmarks. Each run of the distribution produces a file *work/benchmarks* that contains one line of performance information for each frame of runtime. Each line contains 19 cells separated by a comma:

1 Number of tiles in current scene cut. These must be loaded on GPU, but do not have to be rendered in case they are not in view.

2-8 Number of tiles in current scene cut depending on their level in quad tree. First cell is level zero, and seventh cell is level six.
9 Number of rendered tiles in this frame.
10-16 Number of rendered tiles in this frame depending on their level in quad tree. First cell is level zero, and seventh is level six.
17 Frame time in milliseconds. This number includes complete frame render excluding user input processing time.
18 GPU memory as a fraction of maximum available.
19 Number of tiles that were not sufficient in current cut, but their improvement was not ready.

Console. Only outputs printed into console are initialization system properties, frame times that exceed 18 milliseconds, and garbage collector debug output [9]. Frame times are printed next to garbage collector logs so we can see when we drop frames due to scene building issues and when the garbage collector is to blame.

Shader files. As mentioned before, there are two exported formats for shader source codes.

File in format $[hash]_num_src_glsl_[name].[frag|vert|comp]$ denotes numbered source code before preprocessor work. This version is necessary because compiler and runtime error refer to line numbers in this file, not the original source to which we have added compilation attributes.

In file $[hash]_out_src_glsl_[name].[frag|vert|comp]$ is the code after preprocessor work for debugging of that specific workflow. Unfortunately, this code is not formatted, and an external formatter needs to be used.

[9] To better understand Java G1 GC logs, see `https://blog.gceasy.io/2016/07/07/understanding-g1-gc-log-format/`

Appendix B

Measurement details

In this chapter are details related to system performance measurements, such as extra graphs or less relevant test machine details.

B.1 Tile generator performance

In table B.1 are real times that tested machine took to generate Earth tiles of depth 4 for different number of threads.

	# of threads	time [min]	relative speed
physical cores	1	121.68	1.00
	2	63.78	0.53
	3	44.16	0.36
	4	34.68	0.29
virtual cores	5	32.40	0.27
	6	31.80	0.26
	7	28.26	0.23
	8	29.04	0.24

Table B.1: Table with real measured values from tile generator testing.

B.2 Earth viewing performance

In this section are graphs for Earth viewing performance for lowest ($Q = 0.025$) and highest ($Q = 5$) quality factors tested.

B.3 Tested computer

Besides basic parameters of the tested computer that were listed in the table 6.1, we might want to know additional properties of the graphics card,

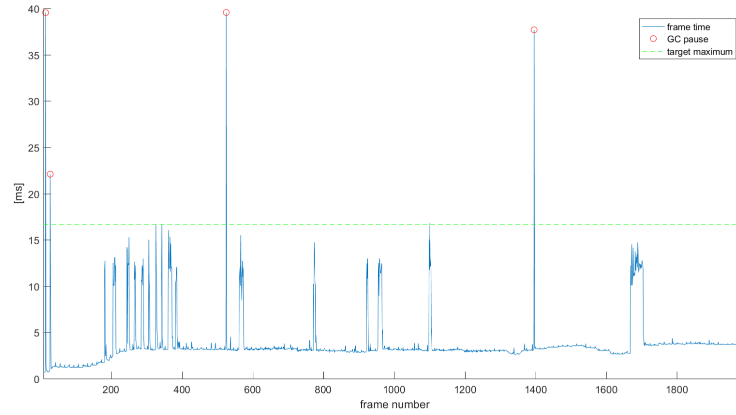


Figure B.1: Frame time for $Q = 0.025$, green line shows target frame time of $16.66ms$, and red circles mark garbage collector pauses.

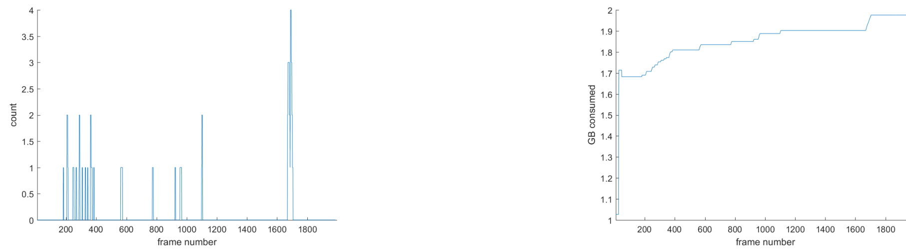


Figure B.2: Number of dissatisfied tiles over time (left), amount of consumed GPU memory in GB (right), both for $Q = 0.025$

and the machine in general. See table B.2 for transfer speeds and graphics card properties.

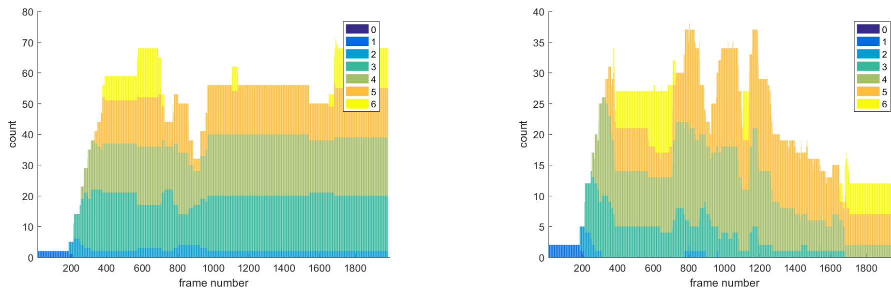


Figure B.3: Current quad tree cut size in number of tiles per depth (left), number of rendered tiles from current cut per depth (right), both for $Q = 0.025$.

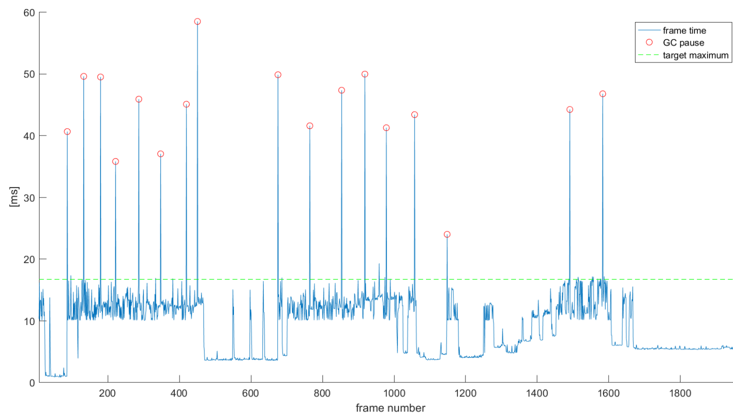


Figure B.4: Frame time for $Q = 5$, green line shows target frame time of $16.66ms$, and red circles mark garbage collector pauses.

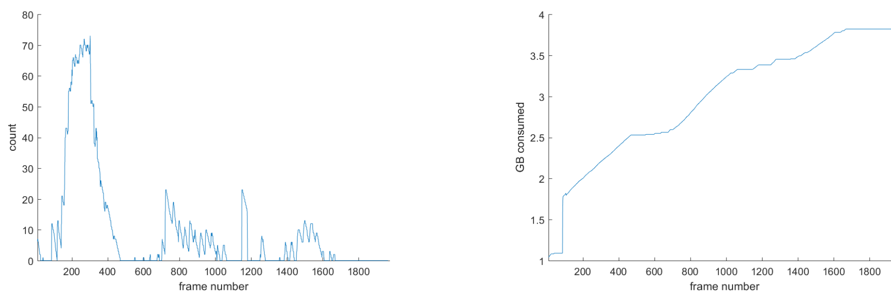


Figure B.5: Number of dissatisfied tiles over time (left), amount of consumed GPU memory in GB (right), both for $Q = 5$

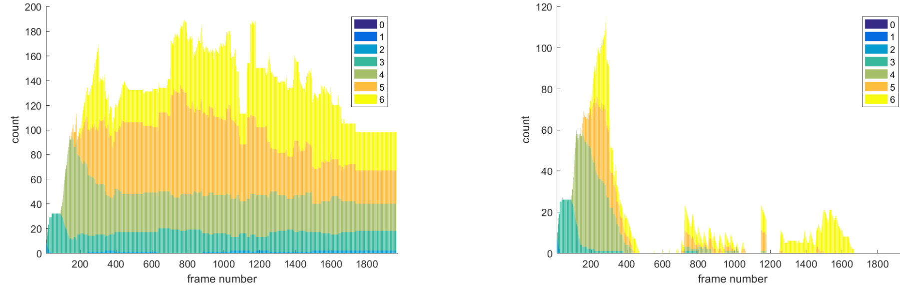


Figure B.6: Current quad tree cut size in number of tiles per depth (left), number of rendered tiles from current cut per depth (right), both for $Q = 5$

Key	Value
MAX_COMBINED_TEXTURE_IMAGE_UNITS	192
MAX_CUBE_MAP_TEXTURE_SIZE	32768
MAX_DRAW_BUFFERS	8
MAX_FRAGMENT_UNIFORM_COMPONENTS	4096
MAX_TEXTURE_IMAGE_UNITS	32
MAX_TEXTURE_SIZE	32768
MAX_VARYING_FLOATS	124
MAX_VERTEX_ATTRIBS	16
MAX_VERTEX_TEXTURE_IMAGE_UNITS	32
MAX_VERTEX_UNIFORM_COMPONENTS	4096
MAX_VIEWPORT_DIMS	32768
MAX_UNIFORM_BUFFER_BINDINGS	84
MAX_UNIFORM_BLOCK_SIZE	65536
MAX_VERTEX_UNIFORM_BLOCKS	14
MAX_FRAGMENT_UNIFORM_BLOCKS	14
MAX_GEOMETRY_UNIFORM_BLOCKS	14
MAX_COMPUTE_WORK_GROUP_INVOCATIONS	1536
MAX_ELEMENTS_INDICES	1048576
MAX_FRAMEBUFFER_WIDTH	32768
MAX_FRAMEBUFFER_HEIGHT	32768
GL_DEPTH_BITS	24
SHADING_LANGUAGE_VERSION	4.50 NVIDIA
VENDOR	NVIDIA Corporation
MAX_COMPUTE_WORK_GROUP_SIZE	[1536, 1024, 64]
MAX_COMPUTE_WORK_GROUP_COUNT	[2147483647, 65535, 65535]
CPU_TO_GPU_NANO_PER_BYTE	0.12541300773620606
DRIVE_TO_CPU_NANO_PER_BYTE	4.45505442475302
CPU_TO_GPU_MB_PER_SECOND	7604.269554017955
DRIVE_TO_CPU_MB_PER_SECOND	214.065693812286

Table B.2: Additional test computer details.

Appendix C

Considered technology

In this chapters are elaborated technologies and engines we looked into.

C.1 Engines

In this section is described process of selection of suitable existing engines for tasks described in the requirements section.

Basic requirements for graphics engine are:

- Engine must be available for all major platforms — Windows, OSX and Linux. This means that it is possible to simply compile source codes for these platforms without too much of additional work.
- Ongoing development of the engine. It is important to make sure the technology will stay with us for a considerable amount of time, and if possible, be improved as well.
- Feature wise complete engine that allows implementation of everything in previous system.
- Active community that will be able to provide support with implementation of more specific features.
- Complete documentation.
- Possibility of GUI integration.
- Access to the source code.

Unity Engine 5. Unity 3D [1] is full feature game engine allowing development on all possible platforms. It prioritizes visual programming unity FlowCanvas [2]) using Unity 3D editor, and does not allow access and modification of source codes of the engine.

It has learning program available, as well as online support for subscription licenses. These are backed by wide community support and community created content on unity asset store.

[1] `unity3d.com`

[2] `https://forum.unity3d.com/threads/flowcanvas-inspired-by-unreal-blueprints.245646/`

According to community lacks newer C# features and *.Net* compatibility, version control causes issues in larger teams, and most importantly, severely limits the ability to make large open worlds without extensive custom architecture and expensive source code access. Unityscript (a version of JavaScript) is supposedly much worse than JavaScript itself.

Flightgear. FlightGear Flight Simulator [3] is complete simulator with an extensive dataset, but due to its license, it is not suitable for commercial products.

Open Scene Graph. Open Scene Graph [4] (OSG) is graphics middleware for virtual scene visualization. It consists of the scene graph that is capable of rendering in real time using OpenGL, loader tools for import of necessary media in the system and so-called node kits, that serve as modules for the base engine.

Amount of created content for this framework and size of the community suggest this is one of the best options, as we have full access to source codes of the rendering process, which is essential for the solution of our precision issues.

In attempts to make the source of OSG work, it turned out that large part of the documentation is outdated and it was difficult to even compile the sources due to inconsistency with dependencies.

Lockheed Martin's Prepar3D. Prepar3D [5] is a flight simulator. It allows the creation of training scenarios for pilots of various vehicles within the system. There is SDK for development for Prepar3D, but it is documentation not very helpful and it seems to be for minor changes and configurations only.

Virtual Battle Space 3. Virtual Battle Space 3 [6] is full feature 3D engine API specifically designed for combat simulators. Because of its compatibility windows-only it cannot be used for our project.

CryEngine 5. CryEngine 5 [7] is a full-feature game engine that allows pretty much everything every other AAA game engine does. It offers visual editor, full access to source codes (but their modification is not allowed under standard license).

According to the community, this engine is somewhat tricky and complicated for use compared to other engines of similar size. It is essential to be able to change source codes of the engine, as one of our particular problems is position precision, which requires such modifications.

[3] `www.flightgear.org`
[4] `www.openscenegraph.org`
[5] `www.prepar3d.com`
[6] `bisimulations.com`
[7] `www.cryengine.com`

Godot engine. Godot engine [8] is a 2D/3D game engine that provides most of the necessary features and is open source so that it could be extended. It goes the way of scripting in a custom python-like scripting language. It has very well structured and complete documentation along with a line of tutorials.

Ogre3D engine. Ogre3D [9] is a fast full-feature 3D game engine. One of the main disadvantages of this engine is its size, as the engine is quite complex and large, and it is difficult to get into, not to mention not all that user-friendly documentation.

Unreal engine 4. Unreal engine 4 [10] is full feature engine for game development. It is pushing interactive visual development using a provided editor and scripting language. The user has full access to source codes of the engine, and it can be modified in full extent.

LWJGL 3. Lightweight Java Game Library [11] is unifying myriad of interfaces under one hood. The only thing it provides is certain assurance of compatibility of already created code with newer versions of wrapped interfaces. This means more freedom for the cost of more time spent programming, but that still may pay off, as it is not required to overcome different design decisions of the engine while implementing special cases of created system.

JOGL. JOGL [12] (Java OpenGL [13]) is, similarly to *LWJGL*, API interface, but it works only with OpenGL. Offers OpenCL (JOCL) and OpenAL (JOAL) bindings as well.

LibGDX. LibGDX is a framework that on the back-end uses *LWJGL*, so it can technically do anything *LWJGL* can do, but it is not necessarily better option at all circumstances. It contains set of helpers for graphics, audio, physics, and math, and it is only up to the user to pick from them. There is no forced design of the system.

JMonkeyEngine. JMonkeyEngine [14] is high level engine built on top of *LWJGL*. It provides all features and constructs of modern game engines, such as *Unity3D* or *UnrealEngine*. Since *jME3* is fully open-source, it is possible to modify any code inside of the engine to fit current needs.

[8] godotengine.org
[9] www.ogre3d.org
[10] www.unrealengine.com
[11] www.lwjgl.org
[12] jogamp.org/jogl/www/
[13] www.opengl.org
[14] jmonkeyengine.org/

	Type	Lang	Activity	Price	License	OS
Unity 5	game engine	C#	ongoing	$125USD/month	per seat	all
Flightgear	simulator	C++	alive May 2016	Free	GNU	all
OSG	scene graph	C++	alive Nov 2015	Free	GNU	all
Prepar3D	simulator	C++	alive Sep 2015	$200USD	per seat	W
VBS3	game engine	C++	alive Jun 2016	NA	NA	W
CryEngine 5	game engine	C++	ongoing	Free	Games only	L/W
Godot	game engine	C++	alive Jul 2016	Free	MIT	all
Ogre3D	game engine	C++	alive Mar 2016	Free	MIT	L/W
Unreal 4	game engine	C++	ongoing	royalties	custom	all
LWJGL 3	graphics API	java	alive Jun 2016	Free	GPL	all
JOGL	graphics API	java	alive Jun 2016	Free	CC	all
LibGDX	graphics API	java	alive Jun 2016	Free	Apache	all
JMonkeyEngine	game engine	java	ongoing	Free	BSD	all
World Wind	Earth view	java	dead Jun 2012	Free	NASA	all

Table C.1: Engine comparison table.

NASA World Wind. NASA World Wind [15] is SDK with an integrated view on Earth with textured surface and various sources of remote content. Because of this project being dead for quite a while, it is not viable as a base for this work, but it can be used as a source of inspiration on how it can be approached, as implemented features in World Wind largely overlap with requirements for our new system.

C.2 Additional technologies and data sources

In this section are listed technologies and data sources that can be coupled to a varying extent with previously listed engines. Use of these should be considered when selecting the new engine.

SIMTHETIQ. Simthetiq [16] is a company providing assets for virtual worlds. Their store contains a wide range of military land, naval and air vehicles. Amount of civilian models is rather small. Simthetiq prides itself on the accuracy of their models, which are supposed to be modeled in several levels of detail, and with rigged moving parts. Advertised environments and airports are not available in their online store. The potential benefit from this model library may be in military scenarios or those few civilian models in air traffic control.

Listed compatibility seems to stem only from available model formats and has no direct connection to engines in question.

Type Model library
Contents detailed military models, scarce civilian models
Activity seems to maintain the store, but not the development. Support is not really necessary in this case

[15] `worldwind.arc.nasa.gov/java/`
[16] `www.simthetiq.com`

Price $2000USD/piece
License redistribution with software [17]
Compatibility CryEngine, Unity 3, OgreEngine, Unreal Engine 3, VBS2, OSG, Source Engine

UnityFS. Unity Flight Simulator [18] is an asset for Unity engine enabling detailed simulation of piloting of an aircraft. Out of the box are available aircrafts to be piloted, but the main feature of this engine is the addition of custom aircrafts without any coding.

This asset could be used as an option to let real pilots control simulated aircrafts, but besides that, it has no other application in our case.

Type Unity engine asset (C#)
Contents aircraft simulation model, 1000 aircrafts
Activity last version (1.04) from February 27, 2014
Price $75USD
License per seat
Compatibility Unity 3.5+ (Guaranteed Unity 5)

BlueSkies. BlueSkies [19] is Unity engine plugin that allows easy import of clouds, fog, wind and precipitation combined with light effects. It also provides accurate star map along with the correct movement of Earth and moon relative to the sun.

Results presented on the official page have somewhat questionable quality, and it could be considered inferior to solutions presented later in this document.

Type Unity engine plugin (C#)
Contents volumetric clouds, fog, wind, lighting, precipitation
Activity Alive at February 2016
Price $2000USD
License per project
Compatibility Unity 5

Sundog Triton. The Triton [20] is software development kit for adding water surface effects and ocean effects into scenes that run either on OpenGL or DirectX.

Allows easy integration in any C++/C# project thanks to pre-compiled libraries and API. This suggests easy integration into any chosen solution, and thus we can conclude this feature as taken care of.

[17] www.simthetiqestore.com/licences.php
[18] unityfs.chris-cheetham.com
[19] agilesrc.com/blueskies
[20] sundog-soft.com/sds/features/ocean-and-water-rendering-with-triton/

Type SDK for OGL and DirectX (C++)
Contents ocean effects
Activity active 3. June 2016
Price $2500USD ($3500USD with source code)
License per project
Compatibility Windows, MacOS, Linux, OpenGL 2.0, OpenGL 3.2+, DirectX9, DirectX11, integration built-in for osgEarth

Sundog Silverlining sky. Silverlining sky [21] is software development kit for adding atmospheric effects in scenes. It not only manages clouds and fog but also night sky with the accurate movement of sun, moon, and stars.

Allows easy integration in any C++/C# project thanks to pre-compiled libraries and API. This suggests easy integration into any chosen solution, and thus we can conclude this feature as taken care of.

Type SDK for OGL and DirectX (C++)
Contents atmospheric effects
Activity active 3. June 2016
Price $2500USD ($3500USD with source code)
License per project
Compatibility Windows, MacOS, Linux, OpenGL 2.0, OpenGL 3.2+, DirectX9, DirectX11, integration built-in for osgEarth

Trian3DBuilder. Trian3DBuilder [22] is an editor for generation of virtual worlds. It allows to generate a database of objects and surfaces within a virtual world and export it in formats compatible with OSG and VBS2.

Since our terrain is mostly generated from real-world data, and we want to keep it that way, this tool is out of the question.

Type world generation tool
Contents visual editor
Activity alive July 7. 2016
Price unavailable
License per seat
Compatibility OSG, VBS2

FullTerrain. FullTerrain [23] is environment upgrade pack for Prepar3D and FSX (Microsoft Flight Simulator X). It consists of global Earth data pack and numerous expansions for countries and airports.

The main issue with this pack is that it is only available in a closed format as *exe* installer, and thus it cannot be used without a support provided by the author.

[21] sundog-soft.com/sds/features/real-time-3d-clouds/
[22] www.triangraphics.de/?q=en/produkte/Trian3d-Builder
[23] fullterrain.com

Type environment library
Contents real world environment model
Activity alive July 5. 2016
Price $1900USD for basic Earth
License commercial end user (redistribution not listed)
Compatibility Prepar3D, FSX

Terra Vista. Terra Vista [24] is terrain generation tool with integrated viewer and export to various engines, from which only notable ones are VBS. VBS solution was rejected, and thus this tool is out of the question as well.

Type generator
Contents terrain generation tool
Activity unknown
Price not listed
License not listed
Compatibility VBS2, VBS3

[24] www.presagis.com/products_services/products/modeling-simulation/content_creation/terra_vista/