



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Analysis of development practices concerning isometric 2D games
Student: Bc. Jan Glaser
Supervisor: Ing. Adam Vesecký
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2020/21

Instructions

The goal of the thesis is to analyze methods of isometric 2D games in terms of development, from the viewpoint of game engines and all aspects that are related to the development of such games. Furthermore, the next output of the thesis will be a working solution concerning the import/export of isometric textures from modeling tools.

Requirements for the environment:

- create two applications
- a prototype for exporting isometric textures
- a prototype of a game that will demonstrate the usability of those textures

Requirements for functionality:

- import and export of isometric textures from a 3D model

Requirements for technological aspects:

- use TypeScript as the main programming language for the prototype of the game
- deploy the solution as a web application

Other requirements:

- all parts will contain automated tests
- design part of the thesis will follow SI methodologies

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 14, 2019



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Analysis of development practices concerning isometric 2D games

Bc. Jan Glaser

Department of Software engineering

Supervisor: Ing. Adam Vesecký

April 12, 2021

Acknowledgements

I would like to express appreciation and thanks to Ing. Adam Vesecký for his advices and consultations during working on this thesis. Without his help and guidance, this work would have never been accomplished. Most importantly, the creation of this thesis would have never been possible without my father, Ing. Vladimír Glaser, Csc., who supported me through my entire study. Thank you very much.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on April 12, 2021

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Jan Glaser. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Glaser, Jan. *Analysis of development practices concerning isometric 2D games*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zaměřuje na analýzu vývoje izometrických 2D her. Důraz je kladen na tvorbu izometrických 2D textur. Součástí práce je aplikace umožňující automatickou tvorbu takovýchto textur z 3D modelovacích nástrojů, a také aplikace demonstrující užití těchto textur. Výstup této práce bude prospěšný herním vývojářům.

Klíčová slova 2D, aplikace, Blender, hra, izometrické zobrazení

Abstract

The primary goal of this thesis is to analyze the methods of isometric 2D games in terms of development. Emphasis is laid on on creation of isometric 2D textures. Part of the thesis is also an application, allowing automatic creation of such textures from 3D modeling tools and even a prototype, demonstrating usage of such textures. The output of this thesis will be beneficial to game developers.

Keywords 2D, application, Blender, game, isometric projection

Contents

Introduction	1
The goal of the thesis	2
1 Introduction to the topic	3
1.1 Game engine	4
1.2 Visual representation of 2D games	5
1.3 Graphical projection	5
1.4 What is an isometric 2D game	7
1.5 Blender	8
1.6 Using 3D modeling tools for the creation of the isometric textures	9
1.7 The problem of creation of the isometric textures	9
2 Survey of existing solutions	11
2.1 Summary	13
3 Realization	15
3.1 Used technology	15
3.2 Exporting a 3D model to 2D images	16
3.3 Processing the Blender output	21
3.4 Other usage of the Spritesheet manager application	27
4 Testing	33
4.1 Blender rendering script	33
4.2 Spritesheet manager	33
5 Game prototype	35
5.1 Used technology	35
5.2 Loading the textures and animation data to a game	36
5.3 Animation JSON	36
5.4 Drawing the textures in a game	38

Conclusion	39
Bibliography	41
A Acronyms	43
B User manual	45
B.1 Minimal requirements	45
B.2 Using an existing 3D model in Blender and exporting to 2D images	47
B.3 Creating a 3D model in Blender and exporting in to 2D images	48
B.4 Creating a floorile texture	52
B.5 Creating a sprite sheet and animation data using Spritesheet manager	54
B.6 Running the game prototype	59
B.7 Playing the game prototype	59
B.8 Running the tests	60
C Content of the data media, provided along with this thesis	65

List of Figures

1.1	An example of pixel art image	3
1.2	An example of a sprite sheet	4
1.3	An overview of the entire sprite sheet creation process	4
1.4	A use case diagram	5
1.5	An example of isometric game scene, simulating the 3D depth	6
1.6	An overview of the projection groups	7
1.7	Examples of graphical projections	7
1.8	An example of a spider, displayed from the side orthographic view	8
1.9	An example of a spider, displayed from the top orthographic view	8
1.10	An example of a spider, displayed in axonometric projection	8
1.11	Screenshot of Blender software	9
2.1	A screenshot of the Spine software	12
2.2	A screenshot of the DragonBones software	12
2.3	A screenshot of the Nima software	13
2.4	Character creation in the Nima software	14
3.1	Camera parented to an empty	17
3.2	Hierarchy of objects in Blender scene	17
3.3	3D scene setup	18
3.4	A panel with properties of the camera	19
3.5	A comparison of a cow model rendered using an orthographic camera and perspective camera in Blender	19
3.6	An example of render output files from Blender	20
3.7	Animation of spider consisting of 6 frames. Only 3 angles (left, right-down, down) are displayed	21
3.8	Output image with a lot of transparent pixels	21
3.9	Common rectangle for two images, used for cutting off the transparent pixels	22
3.10	Monster drawn using the upper-left corner as an origin point	24

3.11	Monster drawn using the calculated origin point	24
3.12	The origin point mark, displayed as a green sphere	25
3.13	Minimal bounding box, highlighted with a white border	25
3.14	The position of the origin point	26
3.15	The final sprite sheet for walk animation of human, facing left	26
3.16	The final sprite sheet for walk animation of human, facing up	27
3.17	Image with-semi transparent pixels	28
3.18	An example of a water splash sprite sheet	28
3.19	Resulting render of a couch	29
3.20	Cropped image of a couch	29
3.21	Resulting isometric floor tile texture	30
3.22	Floor tile sprite sheet used in Diablo 2	30
3.23	Isometric tile in Blender	31
4.1	The result of the Blender render script's automated tests	33
4.2	An example of passing automated tests	34
5.1	UML diagram of a system using component architecture	36
5.2	Drawn monster using the calculated origin point	36
5.3	An example of a json file for model facing in 8 directions	37
5.4	An example of a json file for model seen from one angle	38
B.1	An error when the web browser does not support WebGL	46
B.2	You can choose an axisting animation from the menu.	47
B.3	The cursor shows where the animation ends. On the right, the end number must equal to the end frame of the animation, in this case 26.	48
B.4	Running the render script	48
B.5	Prepared template.blend scene	49
B.6	Created model must be facing left	50
B.7	Moving the camera by changing the Z position	51
B.8	Checking the final monster size for a game	51
B.9	Creating a new animation action	52
B.10	A dropdown menu with available materials for a floor tile	53
B.11	Rendering a floor tile	53
B.12	Saving the rendered file as an image to a disc	54
B.13	Dragging the input images onto to the Spritesheet manager application	55
B.14	Spritesheet manager application interface	56
B.15	The Automatic angle processing tab	57
B.16	Final sprite sheet for non-isometric animation	58
B.17	The terminal output, when running the game prototype	59
B.18	The game prototype, when navigating to the localhost in a web browser	60

B.19 Using exec function to compile a script and run it	61
B.20 Running a script in Blender	61
B.21 Showing a console window in Blender	62
B.22 Tests tab	63
B.23 Tester interface	63

Introduction

Games have been a part of people's lives since time immemorial. Over time, they also found themselves in the field of information technology. Nowadays, we can find many companies that develop video games for profit. The first video game was called *Tennis for Two* and was released in the year 1948. However, the game Pong was the first one that made it to the mainstream and was released in 1972 by Atari company [1][p. 54].

There are usually some moving or stationary objects in a game, which need to be displayed to the user on the computer screen. For displaying these objects, textures and 2D imagery is used. Therefore, to create a game, textures and animations are required.

Animations are traditionally created using the stop-motion technique, where the displayed object is physically manipulated in small increments between the photographed frames. Later, when the frames are played back in fast succession, it appears to an observer that the object is moving. In the game industry, a large image, called a sprite sheet, containing all the mentioned frames is used, and a computer draws the individual frames with some time delay.

Such a frame is called a sprite. Sprite is an image, which in a game can be used as a 2D object in the world space, and therefore can be moved, during a game. Texture is also an image, but is used to change an appearance of a game object (for example background). However, it is not a game object and can not be physically moved in a game.

If we would like to create sprite sheet textures for a monster or game effect, it could be achieved by manually drawing them in some image editor, such as Photoshop. Such a way would be very time-consuming, and considering that we want to have the same animation for all the different directions the character can be facing (left, right, etc.), it would probably not even be possible to draw it manually. However, if such a model would be created in 3D software, then using a script, it can be rendered and converted into 2D sprite sheets.

This thesis discusses the automated creation of 2D textures, which are

created from 3D models. Blender was used as a 3D modeling tool.

The goal of the thesis

This thesis aims to analyze methods used in isometric 2D games in terms of development and all aspects that are related to the development of such games. The output of this thesis will be beneficial to game developers, because it allows automatic creation of 2D animation sprite sheets from 3D model. Furthermore, the next output of the thesis will be a prototype concerning the import/export of isometric textures from modeling tools. The first part of the thesis introduces the reader to the topic. The next chapter deals with the survey of existing solutions. Another chapter follows, describing the realization. The next chapter focuses on testing the output of this thesis. The last chapter is dedicated to the game prototype [2, p. 20], which is also an output of this thesis.

Introduction to the topic

The output of this thesis are applications that create 2D sprite sheets from a 3D model. Such sprite sheets can be used in 2D isometric games. These textures are generated from a 3D model created in Blender. However, the system can be used for creating textures for any 2D game, not only an isometric one.

Let's imagine that we would like to create textures for some monster or game effect, such as the sprite sheet displayed in figures 3.16 or 3.15. Drawing such textures manually would be very time-consuming, especially for all the different directions where the character might be facing (left, right, etc.). However, if such a model would be created in 3D software, then, by using a script, we could render it into separate images, and later on, we could create one large sprite sheet. An example of a sprite sheet can be seen in figure 1.2. Also, the 3D model can be rotated adequately to be facing the requested direction. An overview of the entire process is depicted in figure 1.3. The use case of the entire system is further depicted by a UML[3] use case diagram 1.4. Nevertheless, for creating pixel art 1.1, it is better to draw it by hand or create using some software like Photoshop, rather than using the rendering method described.

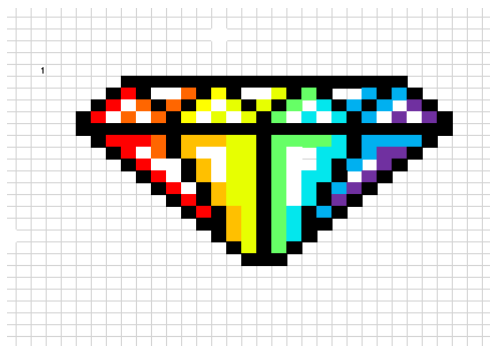


Figure 1.1: An example of a pixel art image[4]



Figure 1.2: An example of a sprite sheet

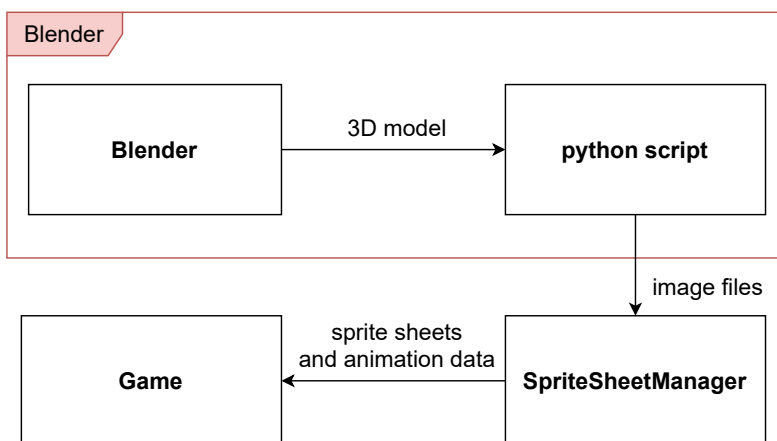


Figure 1.3: An overview of the entire sprite sheet creation process

1.1 Game engine

The term *"game engine"* arose around the year 1990 and referred to first-person shooter games (FPS), such as Doom by id Software [5, p. 30]. It is an environment designed for developers to build video games. Usually, the game engine's core functionalities may include sound, animation, physics, or networking (which are traditionally common for most games), and therefore, the game developer does not have to implement these functionalities by himself/herself [6, p. 10].

Game engines can be divided into heavy game engines, light game engines, toolkits, or libraries.

Heavy game engines, such as Unreal Engine, Unity, or CryEngine, offer the most functionalities, and it is possible to create big games in a short time. On the other hand, with libraries, the developer is provided with much fewer functionalities and has to write most of the game-related code by himself/herself, making the development much more demanding and time-consuming.

It depends on each game developer or company, which of the game engines they choose.

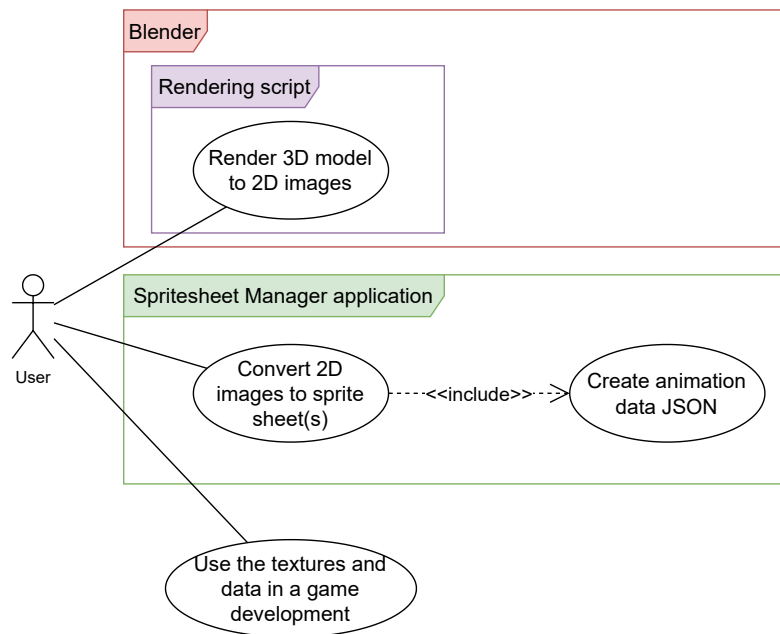


Figure 1.4: A use case diagram, depicting the use case of the applications

1.2 Visual representation of 2D games

There are various visual representations for 2D games to be found. One of the most common ones is the top-down projection, where the user views the world from the above (bird's eye) 1.8, or side-scroller, where the user sees the world from the side 1.9 [7, p. 37]. Both of these views are using an orthogonal projection. Another form of the visualization, which is commonly used in games, is 2.5D (also called pseudo-3D), which allows simulating the 3D depth on the final 2D screen. An example of such a visualization can be seen in figure 1.5.

1.3 Graphical projection

Projection is a process that allows displaying a 3D object onto a 2D surface [8][p. 12]. There are many available graphical projections such as perspective, axonometric or oblique [9, p. 42]. For the sake of the thesis, we are interested in isometric projection, which belongs to the axonometric projection group. However, perspective projection can be used as well. An overview of the projection groups can be seen in figure 1.6. A visualization of some graphical projections is in figure 1.7.

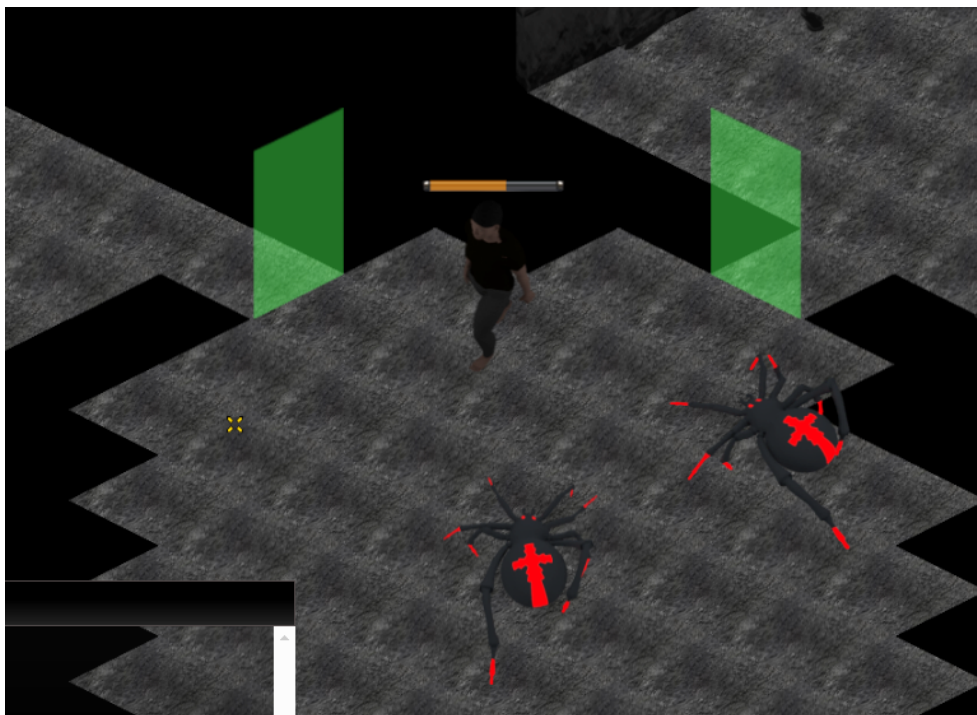


Figure 1.5: Isometric game scene, simulating the 3D depth

1.3.1 Isometric projection

Let's imagine a cube in a 3D space. If such a cube would be viewed using an isometric projection, then there would be an angle of 120 degrees [9, p. 42] between the projected X, Y, and Z axes, as depicted by diagram 1.7.

1.3.2 Dimetric projection

In a dimetric projection, the angle between any two axes (after the projection has been applied) is identical [9, p. 42]. An example of such a projection can be seen in diagram 1.7.

1.3.3 Trimetric projection

In the trimetric projection, all three axes appear to have different lengths after the projection. This means that all of the three angles between respective axes are different [9, p. 42]. An example of such a projection can be seen in diagram 1.7.

As can be observed from the diagrams, all of the axonometric projections allow simulating the 3D depth of the graphics in the final 2D image.

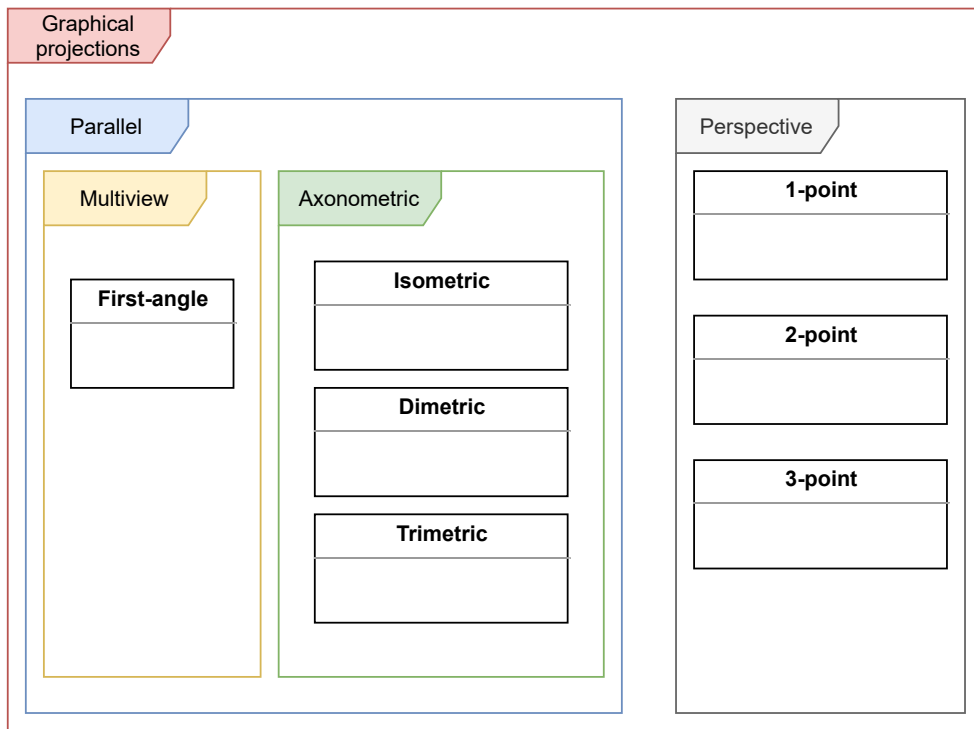


Figure 1.6: An overview of the projection groups

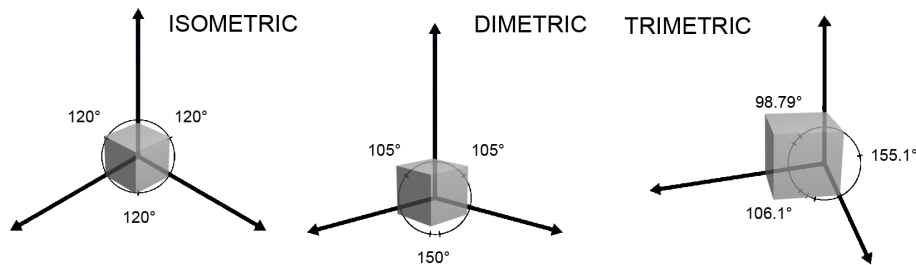


Figure 1.7: Examples of graphical projections

1.4 What is an isometric 2D game

Despite the name, the textures and graphics in an isometric 2D game do not have to be isometric. The game can be using any axonometric projection (as described in the previous chapter) to display the graphics. An example of a 3D spider displayed using an axonometric projection is shown in figure 1.10. All the following game scenes and graphics in this thesis will be described from the isometric projection point of view.



Figure 1.8: Orthogonal projection (side)



Figure 1.9: Orthogonal projection (top-down)



Figure 1.10: Axonometric projection

1.5 Blender

For the 3D modeling tool, I chose Blender, which is free, open-source software, which allows the creation of 3D models. The software is widely used by students, developers, and even CG professionals [10][p. 17], and provides all the basic features, such as modeling, sculpting, or animating. There are many other 3D modeling tools, such as Autodesk Maya or ZBrush. However, these are commercial software.

From this thesis's perspective, the essential features are 3D modeling, sculpting [10][p. 31], texturing, and any features connected to the 3D model creation process. The software also allows to automate most of the actions, which can be achieved by writing scripts in Python language. An example of

1.6. Using 3D modeling tools for the creation of the isometric textures

the Blender software can be seen in figure 1.11 and can be downloaded from [11].

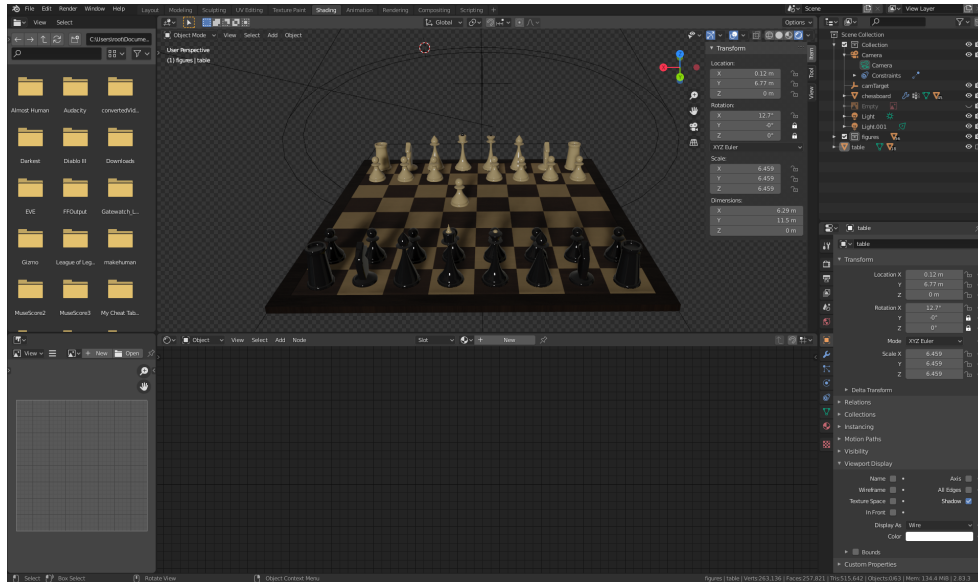


Figure 1.11: Screenshot of Blender software. In this example, with a model of a chessboard.

1.6 Using 3D modeling tools for the creation of the isometric textures

Suppose that we want to create an image of a spider, which can be used in an isometric 2D world. For such a task, a 3D modeling tool, such as Blender, can be used. We need to create the spider model, texture it, and position the camera within the 3D modeling tool to match isometric projection (as displayed in figure 3.3). Then, we can render the final image (as can be seen in figure 1.10).

1.7 The problem of creation of the isometric textures

Imagine the following 2D isometric scene 1.5. Focusing on the spider, we can see that each one is rotated in a different direction. This rotation can not be achieved by rotating the drawn 2D textures, but we must have a texture set for all possible directions. This traditionally applies to any non-stationary object in an isometric game world.

Survey of existing solutions

This chapter focuses on exploring existing solutions that ease up the animation or the texture creation for a game development.

Traditionally the game developers created their own tools for creating 2D textures and graphics. For example, for Diablo 2, most of the in-game and cinematic art was constructed and rendered in 3D Studio Max software. The 2D textures and GUI elements were created primarily using Photoshop [12, p. 2].

According to the popularity list [13] of software for 2D skeletal animations, as of the year 2021, Spine is ranked as the most popular, and behind it is DragonBones software.

2.0.1 Spine

Spine is a 2D skeletal animation commercial software for games. It allows to define a skeleton for a 2D model (in 2D space only), and then, by moving these bones, the character is animated. The final animation data can be exported and used in a game. However, the resulting graphics are not using an isometric projection but an orthogonal one.

The main problem with this approach is that the parts of the character (arms, legs, body, head) still have to be created (usually by drawing them manually). Furthermore, if compared to a 3D modeling tool, such as Blender, if we render a 3D model to a 2D image, we have much more realism and 3D depth. An example of the Spine software can be seen in figure 2.1. The website where Spine software can be accessed can be found at [14].

2. SURVEY OF EXISTING SOLUTIONS

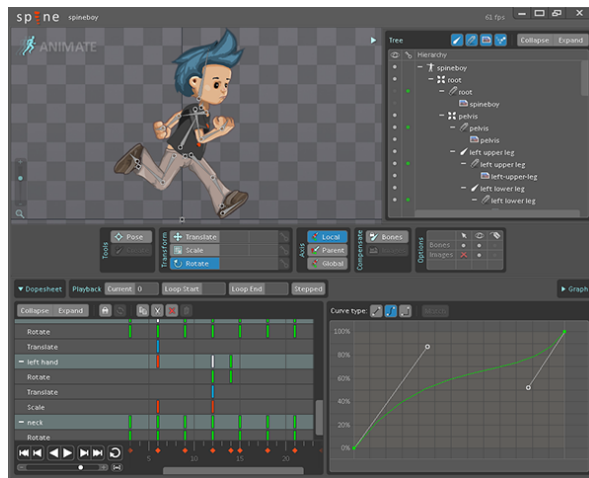


Figure 2.1: A screenshot of the Spine software [15]

2.0.2 DragonBones

DragonBones is a free, open source software, which allows creation of 2D skeletal animations for games. It provides similar features as the Spine software, and it also allows importing imagery and animation data from other software, such as Cocos, or Spine. A screenshot of the DragonBones software is visible in figure 2.2. The software can be downloaded from [16].

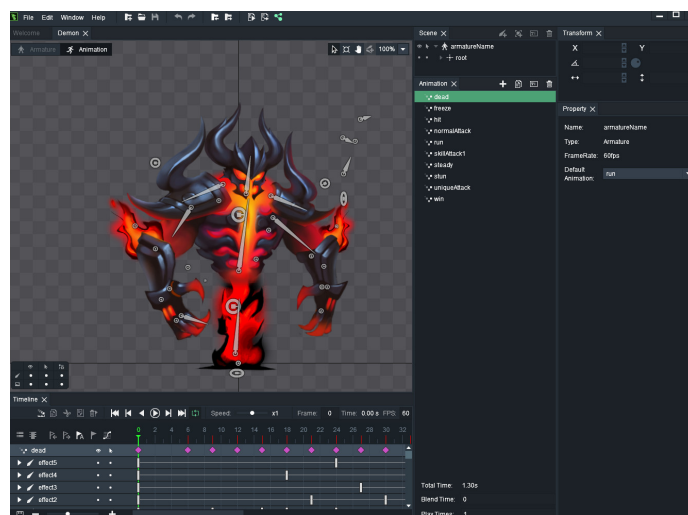


Figure 2.2: A screenshot of the DragonBones software [17]

2.0.3 Nima

Nima is a 2D animation software. It uses a skeletal system to animate the characters, allows dynamic manipulation with the created characters, and runs in real-time. It can run in a web browser, making it very easy to use. However, it is not suitable for creating isometric graphics. An example of a character with a skeleton, created in the Nima software, can be seen in figure 2.4, and the screenshot of the software is in figure 2.3.

The website where Nima can be accessed can be found at [18].

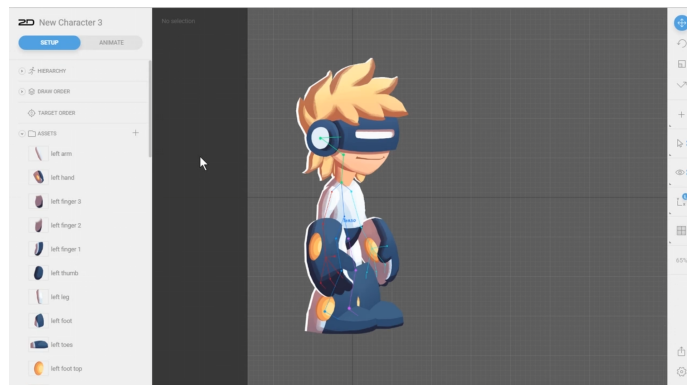


Figure 2.3: A screenshot of the Nima software.

2.1 Summary

I have not found any application, nor a Blender module, which would allow the creation of 2D isometric animations and the creation of sprite sheets for characters, which can be seen from different angles.

2. SURVEY OF EXISTING SOLUTIONS

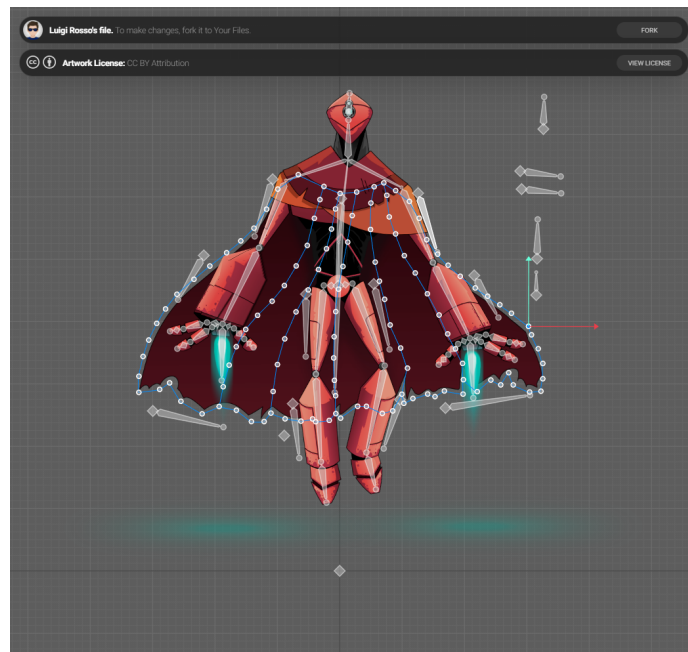


Figure 2.4: Character creation in the Nima software. The character was created by Luigi Rosso.

Realization

The entire project is divided into multiple sub-projects. The first one is the script, allowing to render the 3D model from Blender to 2D images. The second one is an application that allows the processing of the Blender's output and creating sprite sheets and JSON files containing animation data. The third one is the game prototype, demonstrating the usage of the created textures.

3.1 Used technology

In order to accomplish the task, knowledge of the following programming languages was required:

HTML, CSS, Typescript

These are the languages necessary for developing the web application, demonstrating the usage of the textures created. Typescript is a programming language and is a type superstructure of JavaScript. It allows easier development of JavaScript applications [19].

Blender

Experience with Blender is required, including modeling, rigging, animating, texturing, and scripting.

Python

Python [20] programming language is in this thesis used for creating scripts in Blender.

C#

This programming language [21] was used for writing the Spritesheet manager application, which is part of the output of this thesis. The reason for choosing C# is that its libraries allow to effectively work with images and pixel data, and also allow to create user-friendly GUI. The language is not dynamically-typed, such as Python, and makes a larger amount of code more maintainable.

3.2 Exporting a 3D model to 2D images

This chapter explains how the 3D model is exported to 2D images and discusses the in-depth view of how the system works and how the scene is set up. For 3D modeling software, I have used Blender. The presented rendering script, which is a part of an output of this thesis, was created for Blender of the latest version, which at the time of writing this thesis is 2.82.

3.2.1 Setting up the Blender scene

A scene must be set up in a particular way to make it work with the written rendering script.

A collection, named `modelCollection`, must be created, containing all objects related to the 3D model itself. Furthermore, in the root of the scene, a sphere object called `originReference` must be created. The camera is set as a child to an empty object. That empty object is then angled and placed at the desired location, which creates the isometric effect when viewing through the camera. The rotation of the empty object can be adjusted to rotate the camera. This situation is depicted in figure 3.1.

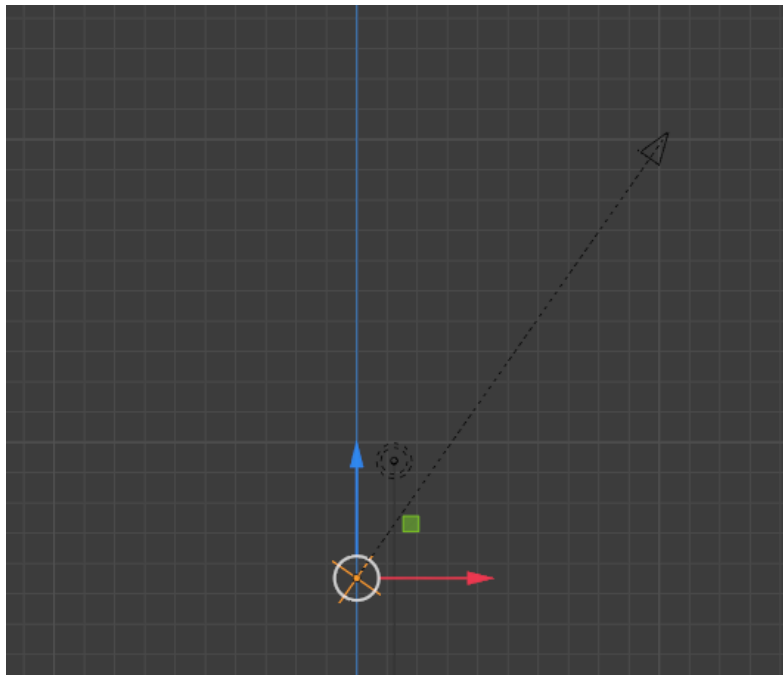


Figure 3.1: Camera parented to an empty. The empty object is displayed using the orange color.

The collection hierarchy tree is captured in image 3.2. In that illustration, a `modelCollection` contains all objects related to the model, which will be rendered. `OriginReference` object is in the scene collection.

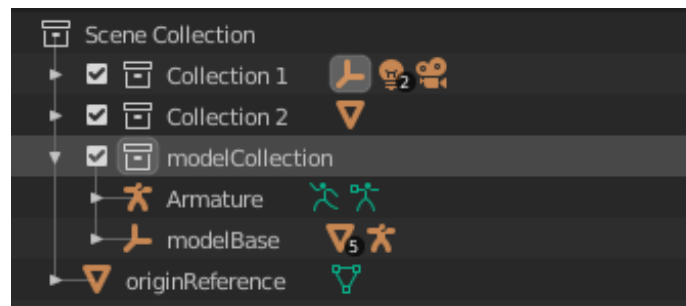


Figure 3.2: Hierarchy of objects in Blender scene. ModelCollection contains all objects related to the model, which will be rendered. OriginReference object is in the scene collection

The 3D scene can be seen in image 3.3.

However, it is not needed to create the scene manually, as a part of this thesis is a template Blender file, which is already set up.

As can be seen, the rotation of the camera can be changed by the user to

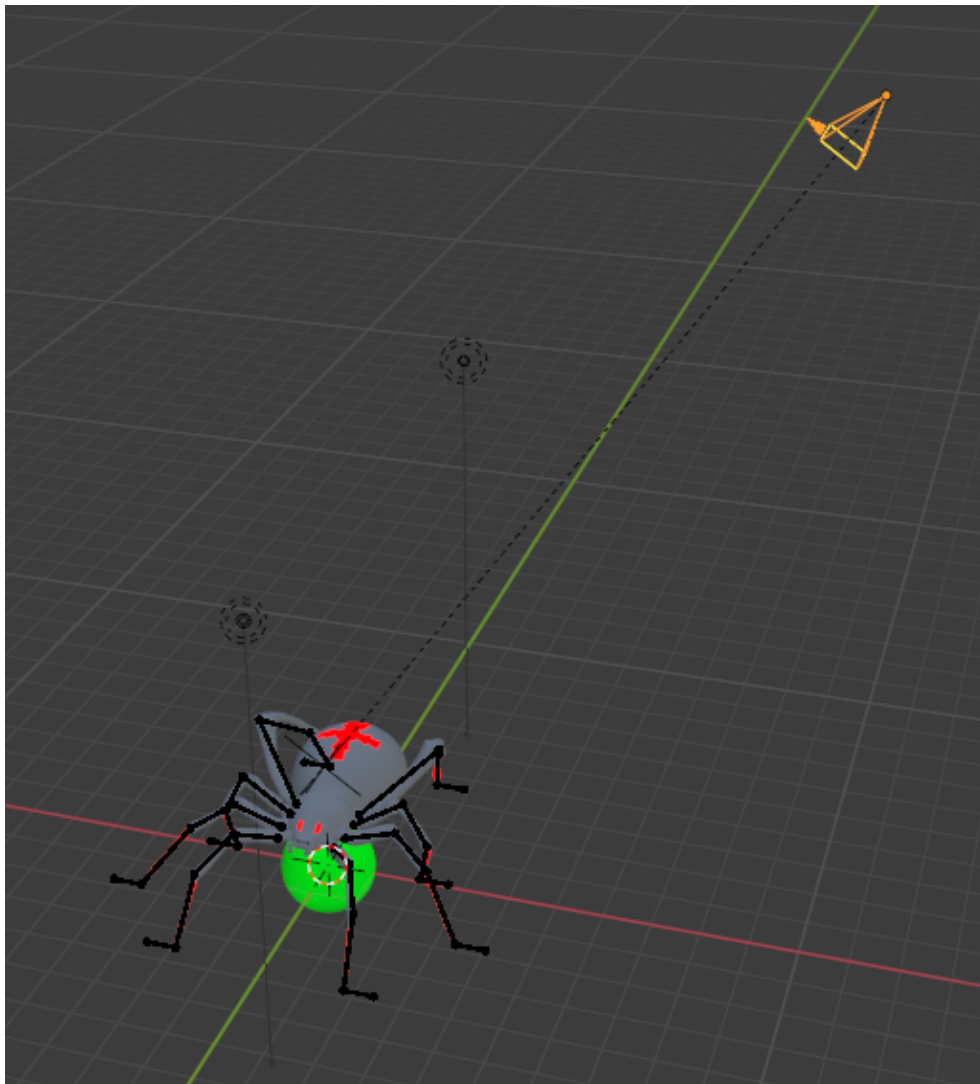


Figure 3.3: 3D scene setup

match the desired final angle of the render. It is important to say that if the camera object is selected, Upon selecting the camera object in the viewport, a camera setup panel becomes available in the Blender GUI, as seen in figure 3.4. In that panel can be seen a type of camera, which, as depicted by figure 3.4 is set to perspective. Users can change it to orthographic. This depends on whether the final graphics will look better using the orthographic camera type or perspective camera type. For the characters (spider, human), a perspective type of camera was used, as the resulting render was better looking.

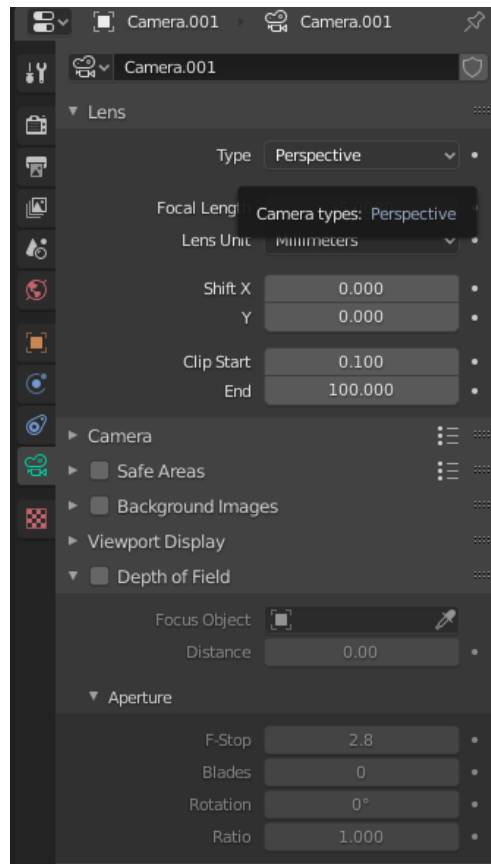


Figure 3.4: A panel with properties of the camera

Orthographic Perspective



Figure 3.5: A comparison of a cow model rendered using an orthographic camera and perspective camera in Blender

3.2.2 The rendering script

The rendering is done using Python script. This script can be found in the gitlab repository [22, 1] or on the data media, provided along with this thesis, in path `\bin\blender\renderAllSides.py`.

The script automatically rotates the root of the camera by a fixed constant angle. Each time, after the rotation, all the animation frames are rendered. As a result, we get a directory with output image files. Each image file contains one animation frame. In illustration 3.6 is a directory with the rendered output, where there is no movement animation of the spider (so each angle has only one frame). There is also an origin point image file. The origin point is discussed in the chapter The origin point in this thesis.

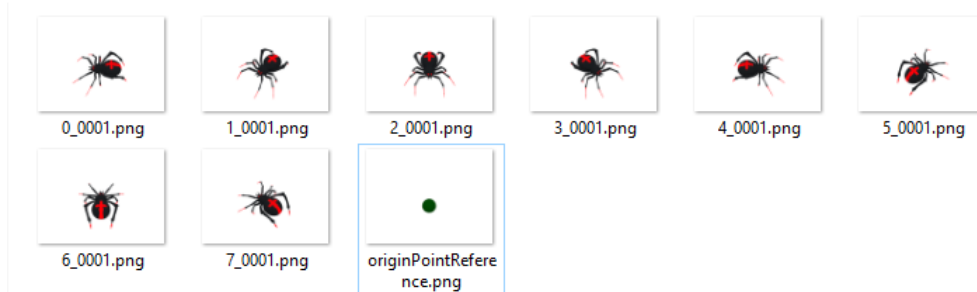


Figure 3.6: Render output files from the Blender

The idea is that in the script is defined in such a way that the character will be rendered from 8 angles. Then, the script renders the current animation of the model, creating output animation frames, and then it rotates the camera root by 45 degrees and renders again. The rotating and rendering actions are repeated until a full turn of 360 degrees is done.

An alternative approach would be to rotate the model instead of the camera root, and the result would be the same. However, this would be a bit more complicated if there were multiple objects from which the 3D model consists, as all the parts would have to be rotated with respect to one shared point in the 3D scene (rotation origin point).

As a result of running the render script, there is a sequence of images for the animation of the model in all eight angles. In the next example output illustration 3.7, there is an animation of a spider, consisting of 6 frames. In the illustration, there are only three angles of the spider (for the sake of the demonstration). Each line is devoted to one rotation angle and contains six frames of an animation for that angle.

Once the script has rendered the model, it will be hidden from the rendering camera. Instead, it would render an origin point sphere. The rendered sphere image is saved to a file called `originReference.png`. The importance of the origin point is discussed in the chapter The origin point.

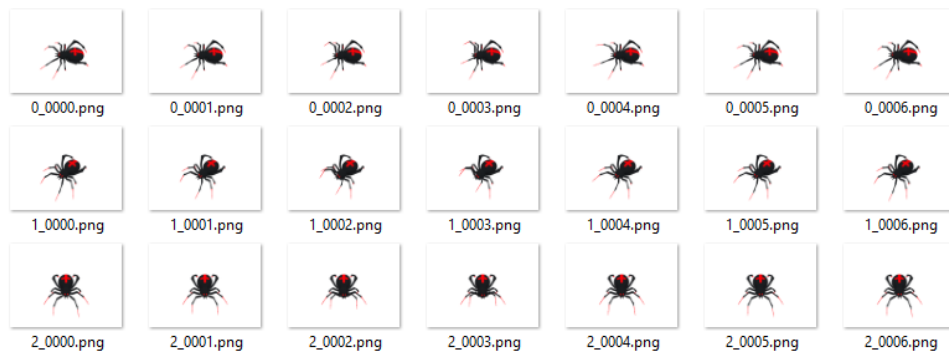


Figure 3.7: Animation of spider consisting of 6 frames. Only 3 angles (left, right-down, down) are displayed

3.3 Processing the Blender output

Having the image sequence rendered from the Blender, if all the images (with the exception of the `originReference.png`) would be placed next to each other in one final image (sprite sheet), it could be used in a game to display the monster animation. However, there are many transparent pixels in each image, and a bounding rectangle can be found to cut off the transparent pixels, and therefore reduce the final size of the sprite sheet. Figure 3.8 illustrates how many transparent pixels are there in one image. The transparent pixels are depicted by green color to make it visible in the illustration.

The application that is part of this thesis's output and allows the processing of the Blender output will be referred to as *Spritesheet manager* from now on.

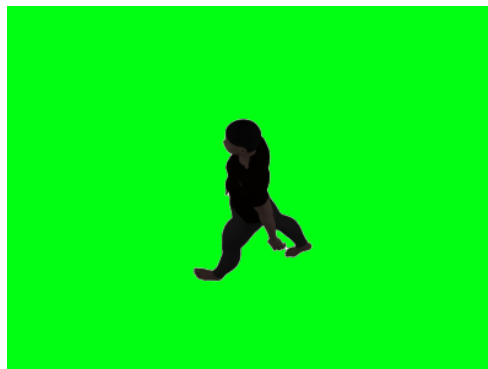


Figure 3.8: Output image with a lot of transparent pixels.

3.3.1 Creating a minified sprite sheet

The task is to find the smallest rectangle, of the same size for all frames. Such a rectangle should cut off as many transparent pixels as possible from all the frames but not any non-transparent pixels.

The algorithm checks for transparent pixels and then creates the final sprite sheets. A sprite sheet for each direction of the character is then created. Each sprite sheet contains sprites for only one direction of the character. This is because traditionally, in a game, the animation (walking left/right, etc.) doesn't change as often as an animation frame. This allows the programmers to load only the needed sprite sheet for a particular direction.

An example of a common rectangle for two frames can be seen in illustration 3.9. This is possible because all the images have the same resolution, thanks to the way how the Blender rendering works.

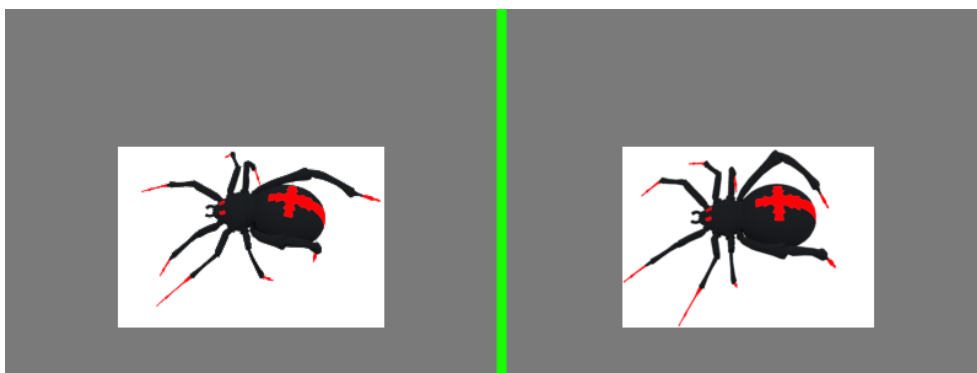


Figure 3.9: Common rectangle for two frames, used for cutting off the transparent pixels. The grey area marks the portion of the images, which is removed, and not used in the final sprite sheet. The green line illustrates where one image ends and the other begins in this illustration.

Now, let's describe the I/O specification for the Spritesheet manager application:

Input

In case of animation of a character, displayed from 8 directions, it is a finite sequence of images of one animation (for example, character walking), displayed from eight directions. The animation directions go in the following order: right, right-up, up, left-up, left, left-down, down, right-down, as defined by the rendering Python script. All the images in such a sequence have the same resolution. Additionally, an image containing the origin reference mark is also an input to the application. In other cases, which form the additional usage of the Spritesheet manager, the input is described in the chapter Other usage of the Spritesheet manager application.

Output

Eight sprite sheet images, where each is representing the character direction described above. A JSON file that contains information about the size of a frame, the amount of frames per line, and the amount of frames in total for each animation angle. The file also includes the origin point, which is described in the chapter The origin point in this thesis.

3.3.2 The algorithm

This chapter describes the algorithm that finds the minimal common rectangle for given image files. Suppose that we want to find the X position of the left side of the final rectangle. The idea is that the algorithm starts at the top-left corner of the image and iterates over all image pixels. The algorithm keeps a variable for remembering the X offset of the left side of the final rectangle. Let's refer to that variable as `X position`. In the beginning, the `X position` is set to the `INTEGER_MAX` value. Once a non-transparent pixel is encountered during the pixel iterating, its X position is saved to the `X position` variable, if it is smaller than the currently found `X position`. After all the pixels were iterated, the `X position` is the desired rectangle's left side's `X position`.

This approach is also used for calculating the top side of the rectangle. For calculating the location of the bottom and right side of the rectangle, the same algorithm is used with an exception, where the `position` variable is initially set to `INTEGER_MIN` value, and during iteration, the algorithm checks whether the acquired position is greater than the one in the variable `position`.

For all input images, this algorithm is used for determining the minimal common rectangle's side's positions. However, for the left and top side, the smallest offset is chosen in the end. For the bottom and right side of the rectangle, the largest offset found among all images is chosen.

This algorithm takes N images, where each has the same resolution of $A \cdot B$. The asymptotic complexity of calculating a minimal common rectangle for given N images is $O(N \cdot A \cdot B)$.

3.3.3 The origin point

This chapter describes what the origin point is and why is it needed. Although the technique above allows automation of processing of any animation desired, it is not enough, and unfortunately, for certain animation sheets, an origin point must be defined in order to make sure that the final graphics are displayed correctly. Let's simplify the problem to only one frame per animation. Imagine we have two sprite sheets for two angles (down and left-up).

If we draw both images in one position, using the upper left corner as an origin, the monster would appear to be standing in a different place, as illustrated by image 3.10.



Figure 3.10: Monster drawn using the upper-left corner as an origin point

However, with correct origin points defined for both images, the monster would appear to be standing in the same position when drawn, as shown by figure 3.11.



Figure 3.11: Monster drawn using the calculated origin point

An origin point is an X, Y coordinate, which is added to the target draw position to shift the resulting graphics in a particular direction.

It is essential to realize that having two sprite sheets, they do not have to have the same size, and therefore, the frame size can differ. This means that the origin point must be calculated separately for each sprite sheet.

3.3.4 Calculating the origin point

The origin point could be defined manually or using a WYSIWYG editor, but it can be calculated automatically. It is worth noting that the exact, precise position is not required, as it would not be notable by the client in the game anyway, thus an approximation suffices.

As discussed earlier, a sphere object is added to the Blender scene. The reason for using a sphere is that it's a 3D object, which will always project as a circle in 2D, no matter the projection (does not matter, where the camera is and at what angle). The origin sphere mark is illustrated by image 3.12.



Figure 3.12: The origin point mark, displayed as a green sphere

When the model is rendered using the automated script, the sphere is hidden from the render. After the model is rendered, the model is hidden, and the sphere is shown. Later, one image containing only the green sphere is rendered.

Now, considering that we find the minimal bounding rectangle for the model image (as seen in illustration 3.13),

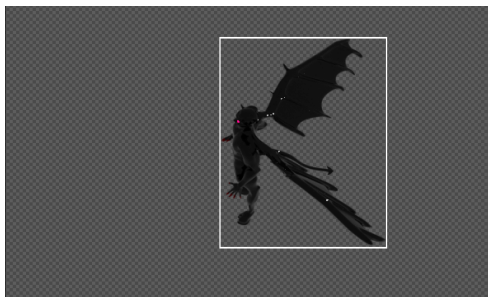


Figure 3.13: Minimal bounding box, highlighted with a white border

3. REALIZATION

also the minimal bounding rectangle for the sphere image is calculated, and its center is calculated to get the center of the sphere, which is the final origin point. Such an origin point's coordinates are relative to the frame size (calculated minimal common rectangle bounding box). The situation is depicted by an illustration 3.14.

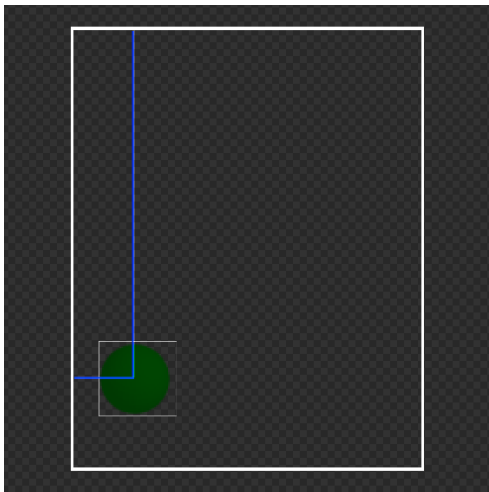


Figure 3.14: The coordinates of the origin point are depicted by blue lines, and the minimal bounding box, displayed using a white line

This is easy to calculate, since all rendered image files have the same resolution. The calculated origin point is saved into a JSON file, which contains the animation data. Such a point must be calculated for each 3D model rotation. The reason is that sprite sheets for different model rotation can have different resolution and frame size, as can be seen in figures 3.15 and 3.16. The first sprite sheet has a resolution of 990 x 417 pixels, and the second has 680 x 510 pixels. Therefore, the calculated origin point will be different for each of these sprite sheets.



Figure 3.15: The final sprite sheet for a walk animation of a human, facing left. Resolution of such a sprite sheet is 990 x 417 pixels

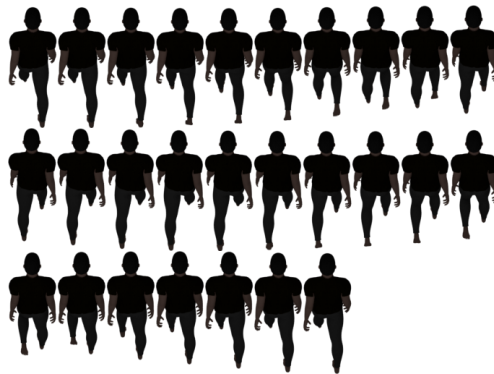


Figure 3.16: The final sprite sheet for walk animation of human, facing up. Resolution of such a sprite sheet is 680 x 510 pixels

3.3.5 Alpha sensitivity lower bound

This chapter explains how the Spritesheet manager application allows to set a threshold for determining transparent pixels. Suppose that a fireball image containing semi-transparent pixels was rendered as a final image from the Blender. Considering RGBA color model, semi-transparent pixels are pixels, which alpha channel is not 255 (fully transparent), but might be, for example, 160, so the pixel has color, but a user can see through it. Thus, it is vital to set the threshold, which is used for determining whether a pixel will be considered transparent by the algorithm. As displayed in figure 3.17, on the left, we can find the input image for the Spritesheet manager. The middle image shows the output when the alpha sensitivity lower bound was set to value of 255. As can be seen, such an output isn't visually pleasing and could not be used in a game. The image on the right displays a result where the value was set to 100. The value 100 is set as a default value in the Spritesheet manager application and works well for a majority of inputs. However, it can be configured by the user as needed. A pixel is considered transparent by the program if its alpha channel value is greater than the alpha sensitivity value. The correct value for particular image inputs must be determined by the user, traditionally by a trial and error.

3.4 Other usage of the Spritesheet manager application

3.4.1 Usage for a classic animation

This chapter discusses how the application and the idea of image processing can be used for other purposes. Suppose that we want to create an animation, which will be viewed from a static angle. An example could be a water

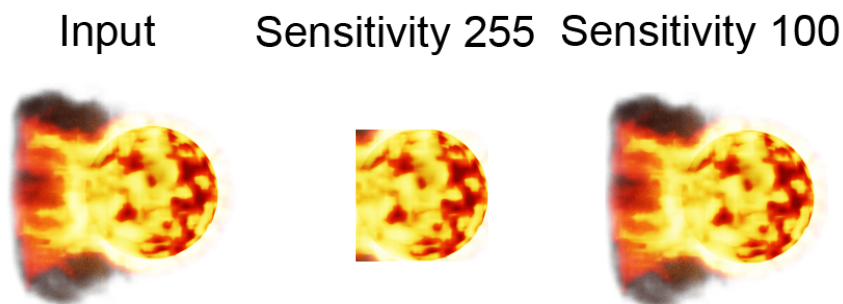


Figure 3.17: Image with semi-transparent pixels. On the left is an input to the Spritesheet manager

splash, which could be seen in figure 3.18. Such an animation can be used, for example, in a naval game, and it is not needed to rotate for eight or more angles.

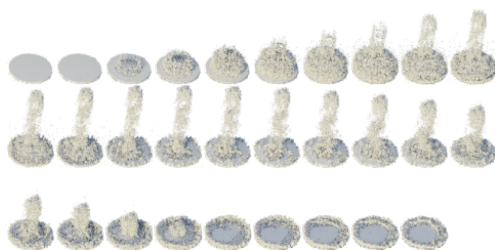


Figure 3.18: An example of a water splash sprite sheet

These textures are called *billboard textures* and are always rotated towards a camera under 90 degrees in a game. In order to create such a sprite sheet, the model in Blender is created first, using a fluid simulation. The fluid is animated, and then the animation is rendered classically through the Blender interface (not using the render script described in this thesis) to produce series of images. Such a series of images is then used as an input for the Spritesheet manager application, which will create only one sprite sheet with a JSON animation data. This process is described more in detail in the user manual's chapter Creating data for any animation seen from only one angle.

3.4.2 Cutting off the transparent pixels for a single image

Suppose that we want to create a couch texture for an isometric game.

Once the model is created and the camera is positioned and angled appropriately, an image is rendered using the Blender interface. A render result can be seen in figure 3.19. In this figure, the transparent pixels are depicted

with a pink color for the sake of the example. As can be seen, there are lots of



Figure 3.19: Resulting render of a couch. Transparent pixels are depicted with pink color for the sake of the example

transparent pixels. The Spritesheet manager application can be used to find the minimal bounding rectangle and cut off the transparent pixels to minimize the image size. The result after cropping is displayed in figure 3.20. The transparent pixels are depicted with pink color for the sake of the example.



Figure 3.20: Cropped image of a couch. Transparent pixels are depicted with pink color for the sake of the example

3.4.3 Creation of a floor tile

This chapter describes how a floor tile texture was created for the game prototype.

Suppose we want to create an isometric tile (as seen in figure 3.21) for a game.



Figure 3.21: Resulting isometric floor tile texture

First of all, a plane is created, and the camera is positioned in a way so that the resulting rendered image would be tileable. For this, I have used sprites from the game Diablo 2 [23, 1] as a reference. In figure 3.22 is an example of a sprite sheet for floor tiles, which was originally used in Diablo 2.

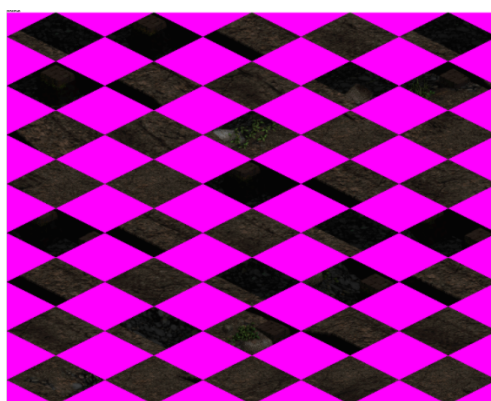


Figure 3.22: Floor tile sprite sheet used in Diablo 2

Then, the task is to position the camera in such a way that the resulting render will have the same shape. The scene can be seen in figure 3.22. The camera angle could be calculated mathematically. However, it would be more time-consuming. Instead of that, I positioned and angled the camera manually, using the sprite sheet as a reference, which was faster.

It is also possible to take the existing sprite sheet from Diablo 2 and edit it in image editing software, such as Photoshop, and place a custom photo of grass or wall over it to create custom floor texture, but keep the tile dimensions. This is a more precise method, but it is more time-consuming, and in the end, even if few pixels are off in the final tile render, it is barely noticeable in the final game.

Once the 3D plane is positioned correctly in the Blender, any tileable texture (material) can be applied to it, in order to create the final tileable floor tile.

3.4. Other usage of the Spritesheet manager application

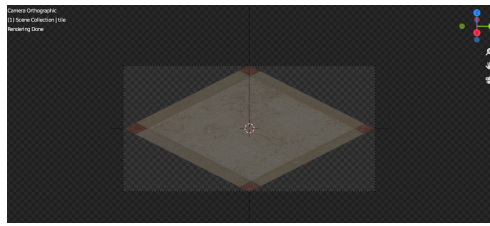


Figure 3.23: Isometric tile in Blender

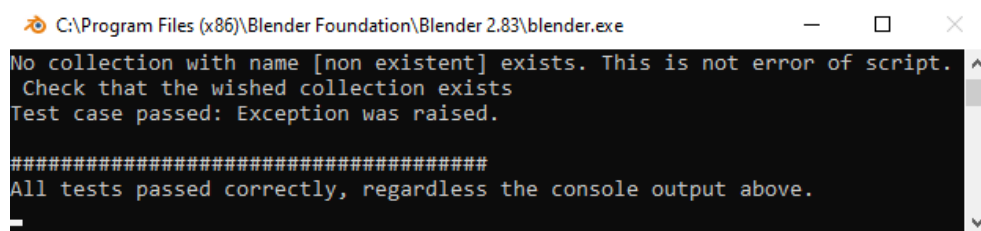
The chapter [Creating a floor tile texture](#) in the user manual explains how to render the floor tile images from the Blender.

Testing

This chapter describes how the created applications were tested to ensure correct behavior. The game prototype was tested manually by playing the game.

4.1 Blender rendering script

The tests for the Blender rendering script test the proper functionality of the `hideRender()` function, which allows to hide or show an object from the render. The next test tests the `hideRenderCollection()` function and its correct behavior when a non-existent collection name is passed as an argument. Because of the nature of the script, the rendering functionality was tested manually by running the script to render various models and then checking the result. A more detailed explanation on how to run the tests can be found in the appendix. The output of running the tests can be seen in figure 4.1.

A screenshot of a Windows command prompt window titled "C:\Program Files (x86)\Blender Foundation\Blender 2.83\blender.exe". The window contains the following text:

```
No collection with name [non existent] exists. This is not error of script.
Check that the wished collection exists
Test case passed: Exception was raised.

#####
All tests passed correctly, regardless the console output above.
```

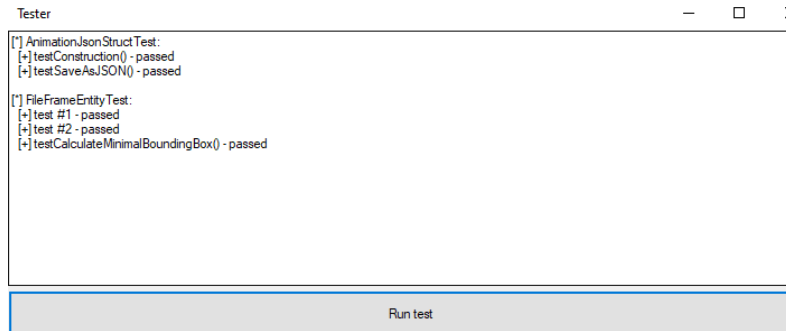
Figure 4.1: The result of the Blender render script's automated tests

4.2 Spritesheet manager

For testing the Spritesheet manager application, automated tests were implemented. These tests are testing the correct creation of a JSON file for

4. TESTING

animation, and also the correct calculation of the minimal bounding box. The proper sprite sheet creation was tested manually by using the application and validating the output. The output of running the tests can be seen in figure 4.2.



```
Tester
[*] Animation.JsonStructTest:
[+] testConstruction() - passed
[+] testSaveAsJSON() - passed

[*] FileFrameEntityTest:
[+] test #1 - passed
[+] test #2 - passed
[+] testCalculateMinimalBoundingBox() - passed

Run test
```

Figure 4.2: An example of passing automated tests

Additional tests can be performed manually by navigating to the directory `.\data\isometricCharacters` in the provided sources. Each subdirectory contains an `input` directory, with input files to the Spritesheet manager and `output` directory, containing an example of expected output. A more detailed explanation of how to work with the application is in the user's manual.

Game prototype

This chapter describes the game prototype web application, which is a part of the output of this thesis. The game prototype demonstrates the usage of created textures. It also demonstrates how the character animation can be switched for all eight angles.

For example, if the main character is supposed to move to the left, then the walk animation (facing left) will be displayed. If the character is stationary, an idle animation is displayed for the character's direction. Depending on the game state, an animation of walk, idle, or attack is chosen and displayed for the character. The world also contains trees, bags, and floor tiles (which can be seen as grass) or a spider. All of these graphics were created using the system described in this thesis. An illustration B.18 shows the game, accessed using a web browser.

5.1 Used technology

For creating the game prototype application, a Typescript programming language was used. Also, the PIXIJS [24] toolkit was used.

PIXIJS is an HTML5 creation engine that already contains some functionalities, which make game development faster [25]. The game prototype is using component-based architecture and messaging system for notifying the game objects. The main objective of such an architecture is to ensure component reusability. A component is a replaceable functionality, which is attached to a simple game object. Because the behaviors are separated from the game object (encapsulated in a component), the entire system is much more maintainable and flexible [26, p. 18]. A UML diagram, depicting a component architecture is in figure 5.1.

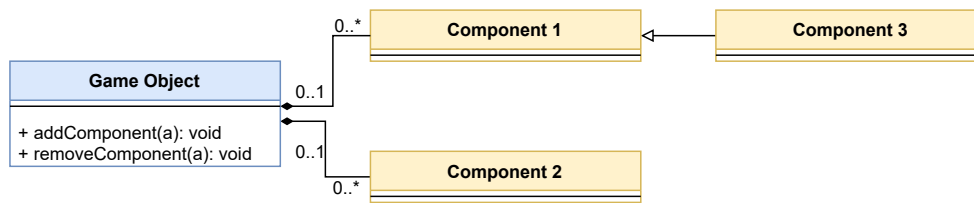


Figure 5.1: A game object can have multiple components attached to it in a component architecture

5.2 Loading the textures and animation data to a game

This chapter describes how the textures and animation data are loaded into the game system. To be able to load and use one animation of a character in a game, a JSON file containing the animation data must be loaded along with all of the texture sheets for all angles. An example where the JSON file is loaded, along with all the sprite sheets (up, down, left, right, up-right, down-right, up-left, down-left), can be seen in illustration 5.2. In that illustration, `this.engine.app.loader` (which is a part of the PixiJS toolkit) is used to load the resources.

```

this.engine.app.loader
.reset()
.add(`player-${Assets.RESOURCE_IDLE}-anim`, './assets/animation/game/map/character/player/idle.json')

.add(`player-${Assets.RESOURCE_IDLE}D`, './assets/img/game/map/character/player/idleD.png')
.add(`player-${Assets.RESOURCE_IDLE}L`, './assets/img/game/map/character/player/idleL.png')
.add(`player-${Assets.RESOURCE_IDLE}D`, './assets/img/game/map/character/player/idleD.png')
.add(`player-${Assets.RESOURCE_IDLE}L`, './assets/img/game/map/character/player/idleL.png')
.add(`player-${Assets.RESOURCE_IDLE}LD`, './assets/img/game/map/character/player/idleLD.png')
.add(`player-${Assets.RESOURCE_IDLE}LU`, './assets/img/game/map/character/player/idleLU.png')
.add(`player-${Assets.RESOURCE_IDLE}R`, './assets/img/game/map/character/player/idleR.png')
.add(`player-${Assets.RESOURCE_IDLE}RD`, './assets/img/game/map/character/player/idleRD.png')
.add(`player-${Assets.RESOURCE_IDLE}RU`, './assets/img/game/map/character/player/idleRU.png')
.add(`player-${Assets.RESOURCE_IDLE}U`, './assets/img/game/map/character/player/idleU.png')
  
```

Figure 5.2: JSON file is loaded, along with all the sprite sheets (up, down, left, right, up-right, down-right, up-left, down-left)

5.3 Animation JSON

This chapter describes the use and structure of the animation JSON file, which is an output of the Spritesheet manager application. There are two variants of such a JSON file. The first one is when the Spritesheet manager is used for processing output from the described Blender rendering script, whereas the model is facing in eight different directions. The other JSON file

structure is created in any other case and is described in chapter Other variant. The attribute which decides, which JSON variant will be created, is the **automatic angle processing** feature available in the Spritesheet manager application. The **automatic angle processing** is a feature that, if enabled, tells the Spritesheet manager application that the output shall be divided into eight sprite sheets, each for one character direction. This is explained more in-depth in user manual's chapter Creating data for isometric characters.

5.3.1 Variant for model facing in 8 directions

Only one JSON file is created for one model animation, and it contains information about all the sprite sheets for all model directions. Each of the model directions is referred to using shortcut letter(s) in the JSON: L - left, R - right, U - up, D - down, LD - left-down, LU - left-up, RD - right-down, RU - right-up.

Here is an example part of the JSON data related to the **left** rotation:

```
1 {
2   "animationSpeed": 100,
3   "L": {
4     "framesTotal": 27,
5     "frameSize": [100, 140],
6     "framesPerLine" :10,
7     "origin":[48,122]
8   },
9   /* The file continues */
```

Figure 5.3: An example of a json file for model facing in 8 directions

An example of the full file is available in the gitlab repository [22, 1] or on the data media, provided along with this thesis, in path

```
\data\isometricCharacters\player walk\output\walk.json.
```

The **animationSpeed** is set with a default value of 100 and can be used for defining time delay (preferably in milliseconds) between each frame of the final animation in the game. This attribute should be defined manually or can be ignored completely. That depends on the final game implementation. The **framesTotal** is the total number of frames of the animation (in the case of this discussed example, for the left rotation of the model). The **frameSize** is the width and height (in pixels) of each frame in the sprite sheet. The **framesPerLine** is the number of frames per line in the sprite sheet, understanding that if an animation consists of 12 frames and ten frames per line in the image, then the last line will contain only two frames. The **origin** denotes the X and Y coordinates of the calculated origin point. This argument, however, might not be present if the origin point was not requested to be calculated by the user.

5.3.2 Other variant

This chapter explains the structure of the animation JSON file, which is created when automatic angle processing is disabled in Spritesheet manager.

Here is an example of the full JSON file:

```
1  {
2  "framesTotal":49,
3  "frameSize": [205, 203],
4  "framesPerLine":10
5  }
```

Figure 5.4: An example of a json file for model seen from one angle

An example of the file is available in the gitlab repository [22, 1] or on the data media, provided along with this thesis, in path `\data\frames to spritesheet\output\output.json`.

5.4 Drawing the textures in a game

To draw a character animation, there must be a system in the game, that chooses, which variation of the animation to display with respect to the character orientation. Having loaded all sprite sheets, we create an animation instance of the class `GraphicsAnimation` for each animation angle. Then, according to the situation (when a character moves), we switch to an appropriate animation for the movement direction.

The `GraphicsAnimation` class takes the animation data as an argument in a constructor. Namely, it is the total amount of frames in the sprite sheet, width and height of one frame (in pixels), number of frames per line in the sprite sheet and the animation speed (in milliseconds). The class instance can be then used for getting the current frame's x and y position, which can be used for drawing the specific frame. The class also allows to automatically advance to the next frame of the animation, based on the elapsed time.

A certain animation frame is changed in `CharacterAnimationBase.ts`, when an appropriate message is received in the `onMessage()` function. The origin point is applied in `Character` class, in function `updateRenderPosition()`.

Conclusion

This thesis aimed to analyze methods of isometric 2D games in terms of development and all aspects related to the development of such games. The thesis's output is a working solution concerning the import/export of isometric textures from modeling tools.

In the thesis, I analyzed the methods of isometric 2D games in terms of development along with a task of creating isometric 2D textures from a 3D modeling tools. Based on that, I designed and implemented three applications, the first one allowing the export of 3D models to 2D images. The second application creates minified sprite sheets from the imagery output and the third application is a game prototype, deployable as a web application, which is demonstrating the usage of created texture sprite sheets. I also implemented automated tests which ensure that the applications are working correctly. This thesis's output will be beneficial to game developers, as it provides automated creation of 2D isometric textures from 3D modeling tools. Therefore, there is no need to manually draw the final sprite sheets but only create a 3D model and later export it to the final sprite sheets.

All the requirements of the thesis's assignment were met.

Bibliography

- [1] Wolf, M. J.: *The video game explosion: a history from PONG to Playstation and beyond*. ABC-CLIO, 2008.
- [2] Aleem, S.; Capretz, L. F.; Ahmed, F.: Game development software engineering process life cycle: a systematic review. *Journal of Software Engineering Research and Development*, ročník 4, č. 1, 2016: s. 1–30, doi: 10.1186/s40411-016-0032-7.
- [3] Pender, T.: *UML bible*. John Wiley & Sons, Inc., 2003.
- [4] An example of pixel art image. <https://www.hugovela.com/daily-directions/pixel-art-an-intro-to-digital-art>, [Online; accessed 25-March-2021].
- [5] Gregory, J.: *Game engine architecture, 3rd Edition*. crc Press, 2018.
- [6] Kelly, C.: *Programming 2D games*. CRC press, 2012.
- [7] Larochelle, A.: A new angle on parallel languages: The contribution of visual arts to a vocabulary of graphical projection in video games. *G/ A/ M/ E Games as Art, Media, Entertainment*, ročník 1, č. 2, 2013.
- [8] Napieralla, J.: Comparing Graphical Projection Methods at High Degrees of Field of View. 2018, diva2: 1229190.
- [9] Camba, J. D.; Otey, J.; Contero, M.; aj.: Design Graphics.
- [10] Mullen, T.: *Mastering blender*. John Wiley & Sons, 2011.
- [11] Blender. <https://www.blender.org/>, [Online; accessed 16-January-2021].
- [12] Schaefer, E.: Postmortem: Blizzard Entertainment's Diablo II. *URL: http://www.gamasutra.com/view/feature/3124/postmortem_blizzards_diablo_ii.php*, 2000.

BIBLIOGRAPHY

- [13] List of the best 2D skeletal animation software. <https://www.slant.co/topics/588/~best-2d-skeletal-animation-tools>, [Online; accessed 25-March-2021].
- [14] Spine. <http://esotericsoftware.com/>, [Online; accessed 16-January-2021].
- [15] Screenshot of the Spine software. <http://fresh-softs-4u.blogspot.com/2015/03/download-esoteric-software-spine-pro-21.html>, [Online; accessed 16-January-2021].
- [16] DragonBones. <http://dragonbones.com/>, [Online; accessed 16-January-2021].
- [17] Screenshot of the DragonBones software. <http://brasilgamedev.blogspot.com/2016/11/crie-animacoes-2d-com-software-gratuito.html>, [Online; accessed 16-January-2021].
- [18] Nima. <https://www.rive.app/>, [Online; accessed 16-January-2021].
- [19] Bierman, G.; Abadi, M.; Torgersen, M.: Understanding typescript. In *European Conference on Object-Oriented Programming*, Springer, 2014, s. 257–281.
- [20] Summerfield, M.: *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2010.
- [21] Perry, S. C.: *Core C# and .NET*. Prentice Hall PTR, 2005.
- [22] Glaser, J.: Gitlab repository with sources. <https://gitlab.com/glaseja1/masterthesisrepo>, 2021, [Online; accessed 16-January-2021].
- [23] Diablo sprites. https://www.sprites-resource.com/pc_computer/diablo2diablo2lordofdestruction/?source=genre, [Online; accessed 16-January-2021].
- [24] PixiJS. <https://www.pixijs.com/>, [Online; accessed 16-January-2021].
- [25] Van der Spuy, R.: *Learn Pixi.js*. Apress, 2015.
- [26] Porter, N.: Component-based game object system. *Carleton University*, 2012.
- [27] Wittern, E.; Suter, P.; Rajagopalan, S.: A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, New York, NY, USA: Association for Computing Machinery, 2016, ISBN 9781450341868, str. 351–361, doi: 10.1145/2901739.2901743.

Acronyms

WYSIWYG What you see is what you get

GUI Graphical User Interface

JSON JavaScript Object Notation. It is a standard file format

URL Uniform Resource Locator

User manual

All the sources discussed can be found in the GitLab repository [22] or on the data media provided along with this thesis. As some of the files in the repository are quite large (such as `.blend` files), it is recommended to clone using `SSH` instead of the `HTTP` protocol, as cloning using the `HTTP` protocol has maximum file size limits and may cause troubles.

B.1 Minimal requirements

This chapter describes the minimal requirements for running all the applications, which are the output of this thesis.

B.1.1 Spritesheet manager

Application Spritesheet manager is written in `C#` for Windows operating system. The binary executable provided is compiled for 64-bit Operating System, and it was tested and worked with on a machine with Windows 10 operating system. If you want to run the binary application on a 32-bit system, you need to compile the binary from sources for the target platform. The source code was written and compiled using Microsoft Visual Studio `C# 2010 Express` environment. To run the provided Spritesheet manager binary application, `.NET Framework` is required. However, it is not always necessary to install these libraries. On a freshly installed operating system, Windows 10, these libraries were already part of the system. However, if needed, the `.NET Framework` libraries can be downloaded from <https://www.microsoft.com/net/download/dotnet-framework-runtime>.

B.1.2 Exporting 3D model to 2D textures using Blender

To be able to export the 3D model to 2D textures, using the way described in this thesis, Blender needs to be installed on a system. It can be downloaded

for free from [11, 1].

It is possible to perform the export from Blender to images on a Linux or Windows operating system, and then for using Spritesheet manager, move to a windows operating system. However, it is recommended to use Windows for all of these actions because you do not need to copy the output images from one system to another.

B.1.3 Game prototype

For running the game prototype application, a machine with Linux operating system is required. Also, an application `npm` is needed. `npm` stands for Node Package Manager, which is a large repository of various JavaScript-based packages [27]. The package manager will figure out all the needed dependencies, and allow to later download all the packages, needed by the project. For installing that program, run the command `apt-get install npm` in the terminal. As the game prototype is a web application, a web browser is needed to play the game. The web browser must support WebGL. Otherwise, the user will see a black screen with an error in the developer console, as displayed in figure B.1. The game was tested with both firefox and chrome browsers. However, chrome seems to be a better choice.

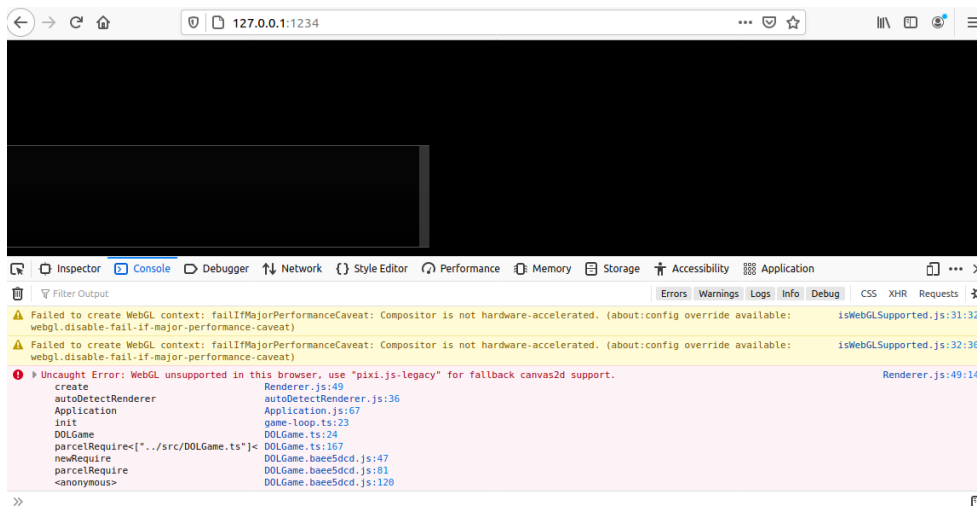


Figure B.1: An error when the web browser does not support webGL

B.2 Using an existing 3D model in Blender and exporting to 2D images

This chapter explains how to use an existing model, provided along with this thesis, to export it to 2D images. Open a provided file with an existing model. In this example, we will use the `.\bin\blender\playerModel*.blend` file. As all the camera positioning, a model position, origin reference object, and sizing is already set up, along with an animated, rigged model, you can directly navigate to the **Animation** tab and make sure that you are in **DopeSheet** and **Action Editor**, as seen in figure B.2.

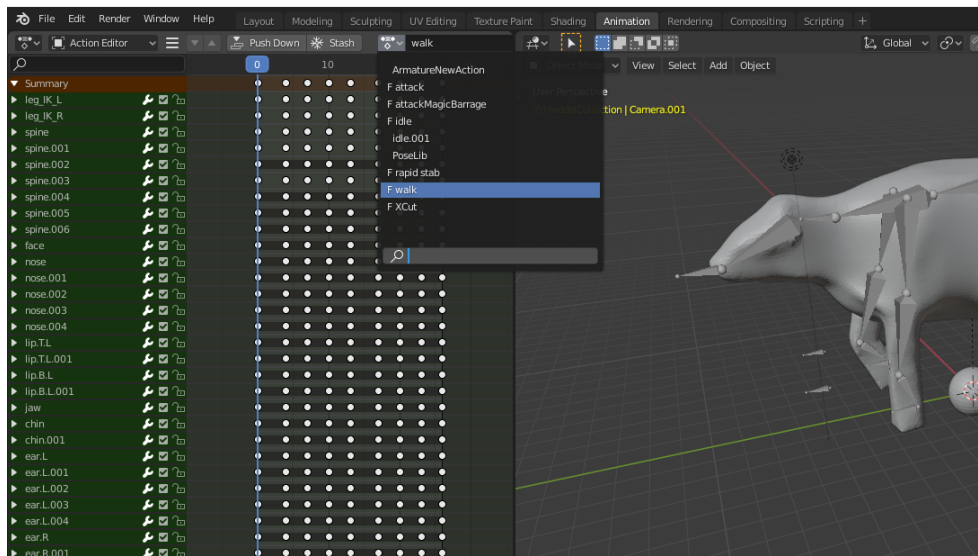


Figure B.2: You can choose an existing animation from the menu.

From the drop down menu, select an animation by name, as displayed in figure B.2. It is important to select an animation, which has **F** in front of the name. In this example, we choose an animation with the name **Walk**. Such an animation can be already selected, however, upon opening the file.

Set the end of the animation properly to the number of frames of your animation, as displayed in figure B.3.

The number of frames can be discovered by moving the cursor to the last set of keyframes, and the cursor will display a number, with the total number of frames of the animation. In the case of the discussed figure, it is number 26. Such a number must be set as the **end** value, which can be seen in the right-bottom part of the figure.

Select the **Scripting** tab, and click on the arrow to run the script, as seen in figure B.4.

Please note that Blender will freeze until the rendering has finished. There seems to be no way to deal with that from the python script, so wait until the

B. USER MANUAL

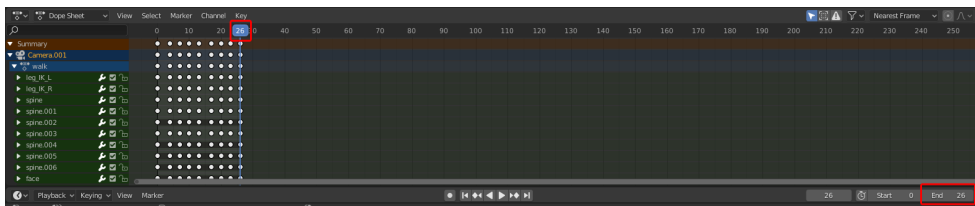


Figure B.3: The cursor shows where the animation ends. On the right, the end number must equal to the end frame of the animation, in this case 26.

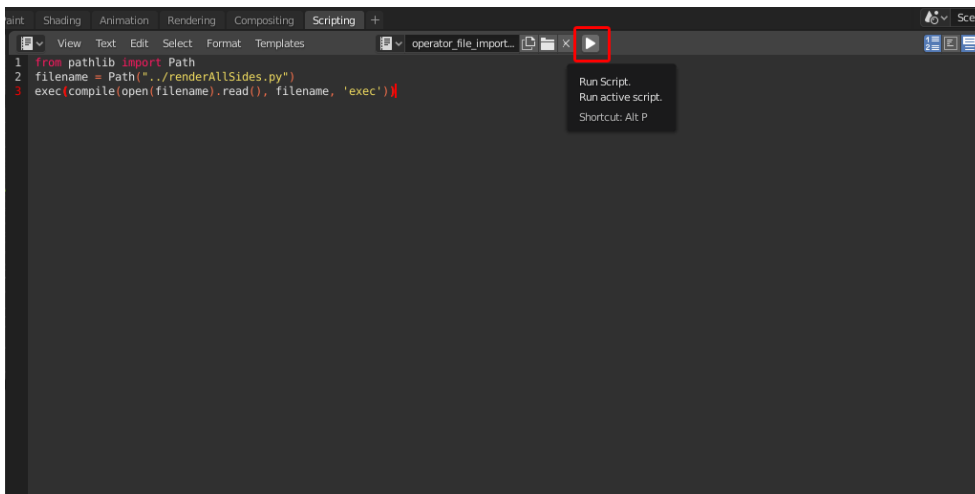


Figure B.4: Running the render script

Blender window unfreezes.

The script will create a directory `C:\tmp` (if not yet exists), and the resulting images will be saved there. While the script is rendering, you may open the output directory and explore the images as they are being created. If you want to change the output directory, you have to open the `renderAllSides.py` script, and at the beginning of the file, change the variable `outDir = str(Path('/tmp'))`; value to the requested output directory. If the output directory does not exist, it is created by the Blender application, which runs the script.

B.3 Creating a 3D model in Blender and exporting in to 2D images

This chapter explains how to create your 3D model, along with all the setup. This chapter expects you to create a 3D model, animate and rig it in Blender, and have some experience with Blender or 3D software. If you only want to try

B.3. Creating a 3D model in Blender and exporting in to 2D images

out the rendering system on an existing 3D model file, navigate to the Using an existing 3D model in Blender and exporting to 2D images chapter instead. To create a new model for a game, navigate to the `.\bin\blender\template` directory in the provided sources, and create a new copy of the `template` directory. The new copy of the `template` directory must be in the `.\bin\blender` directory so that the relative path to the rendering script is valid.

Upon opening the copied `template.blend` file, you will see a scene with a cube, as displayed by figure B.5.

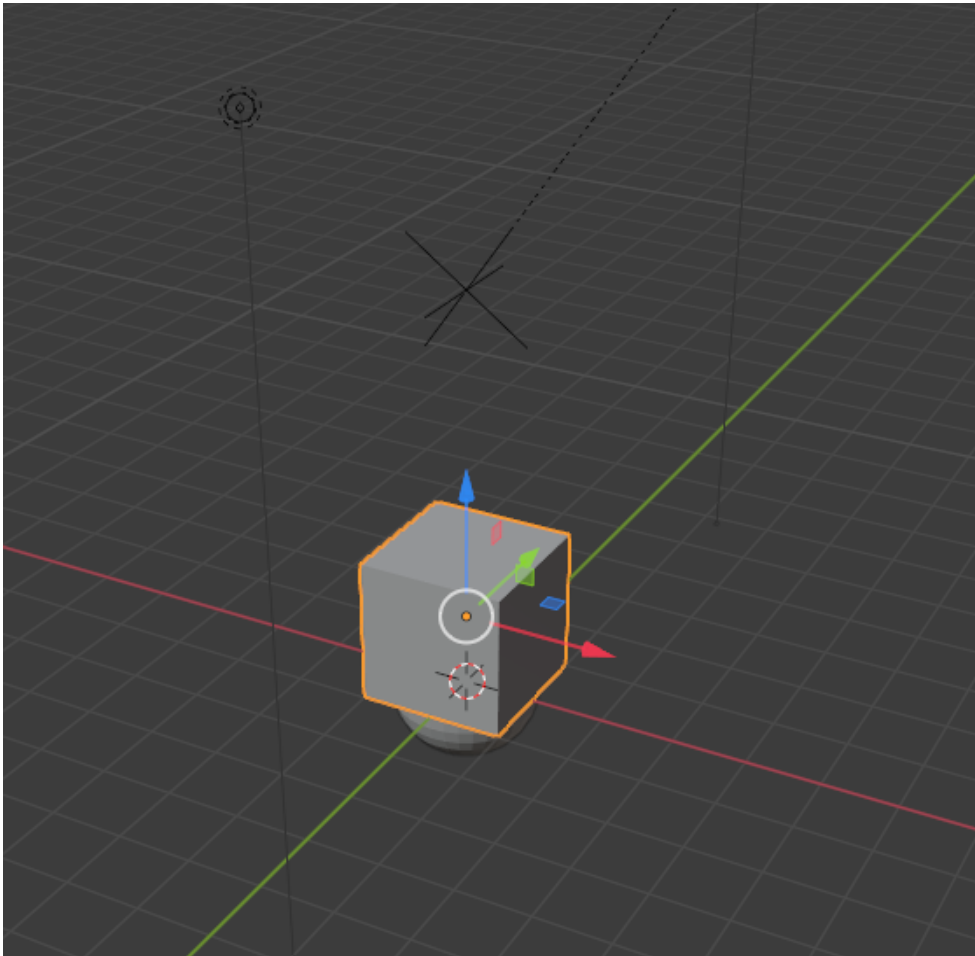


Figure B.5: Prepared `template.blend` scene

Delete the cube object, and create, texture, and rig your own model. Enter camera perspective mode by hitting the `0` key on a Numpad. The created model must be facing left, as shown in figure B.6. Also, your model must be inside the `modelCollection` collection. In the illustration, the model cow is a child of the `ArmatureNew`, which is in the `modelCollection`. The `template.blend` file also contains `originReference` object, which is a sphere

with green material. Please do not delete the object and keep it in the scene. Also, change its position to mark the position of the feet of the model. The material of the object is not important. The only important thing is that the `originReference` object does not have transparent material.

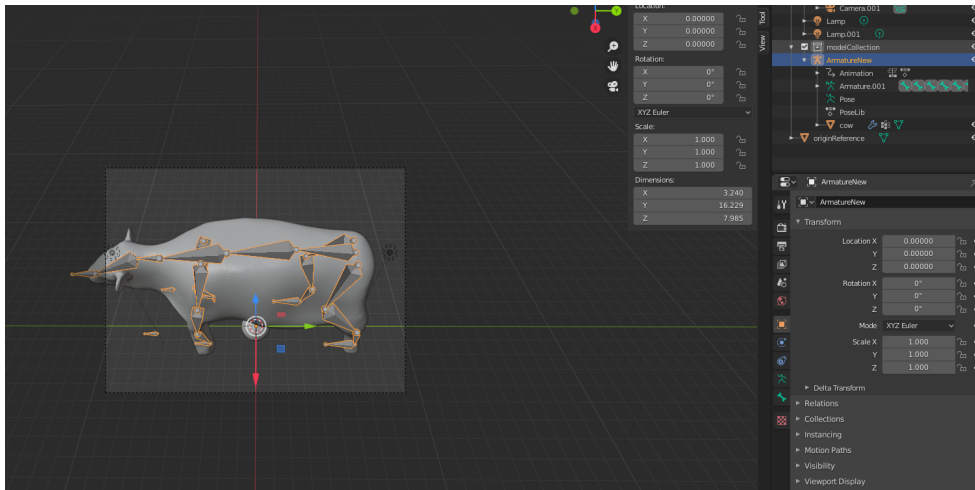


Figure B.6: Created model must be facing left

If the model does not fit the camera view (as in figure B.6), move the camera further away (or closer) by changing the Z position of the camera object, as shown in figure B.7.

Press the F12 key to preview the render. This allows you to see the result you got so far. The rendered output reflects how large the monster image will be in your final game (not considering the transparent pixels). This can be seen in figure B.8.

If you want to make the monster smaller or larger, there is no need to scale the model. You can only change the render resolution percentage, which can be seen on the right in the figure.

Select the **Animation** tab, and make sure that you are in **DopeSheet** and **Action Editor** as seen in figure B.2. Create a new action by clicking on the button displayed in the figure B.9.

Then, animate your model. Make sure that the animation is linked to the model's armature before rendering. This is important in the case of having more animation actions for one model. Then, every time before rendering, you need to select the animation from the action drop-down menu. Now having the model, you can follow the tutorial described in the chapter Using an existing 3D model in Blender and exporting to 2D images.

B.3. Creating a 3D model in Blender and exporting in to 2D images

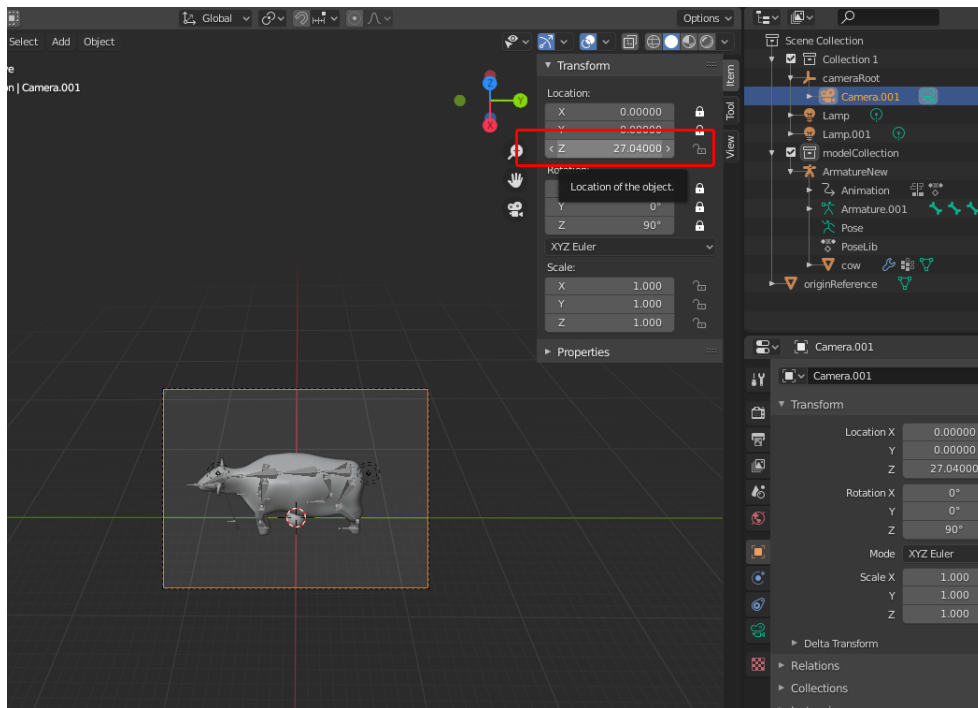


Figure B.7: Moving the camera by changing the Z position

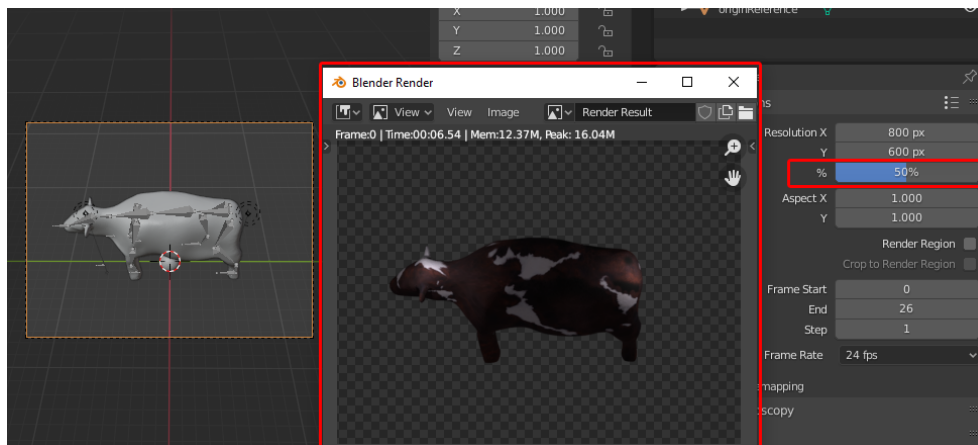


Figure B.8: Checking the final monster size for a game. On the right can be seen the resolution percentage, which is 50 percent

B.3.1 Expected limitations

This chapter discusses the expected limitations for the rendering script used in the Blender. The script's main objective is to automate compiled render process and is not created for dealing with all possible scenarios, so it is

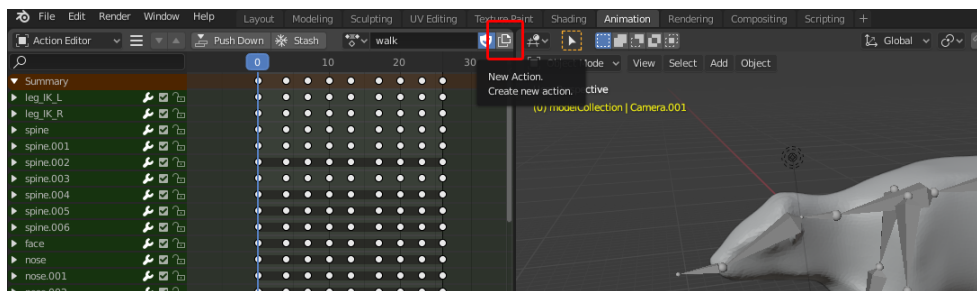


Figure B.9: Creating a new animation action

possible that the script can fail if the user does not follow the manual precisely. As there are many possible situations that can happen in Blender scene setup, it is impossible to check for all of them, such as when the character model, which we want to render, would be a child of the rotated `cameraRoot` object. In such a case, the rotation during the rendering process would cancel out, and the resulting 2D image of the model would always be facing in only one direction. To use the script without complications, it is vital to follow the instructions in the user manual and, ideally, use the provided `template.blend` file, as described in the instructions.

B.4 Creating a floorile texture

This chapter explains how the floor tile texture can be created. Creation of the floor tile results in having one tileable image, as displayed in figure 3.21. Open the provided Blender file in `.\data\floortile\floortiles.blend`. In the opened window (Shader editor), select from the drop-down menu any available material, as displayed in figure B.10.

Hit the `F12` key for rendering, and you will see a window with the final image, as displayed in figure B.11.

The image is in the final real size and does not have to be resized when drawn in a game. From the top menu, choose `Image` and `Save as` to save the render as an image, as displayed in figure B.12.

Then, it is only the matter of the game implementation, on how to draw these textures. It is also a good idea to use the Spritesheet manager application to merge multiple floor tile images into one sprite sheet so that the game client would load one file instead of more files. It is not needed to cut off any transparent pixels from the floor tile image.

B.4. Creating a floorile texture

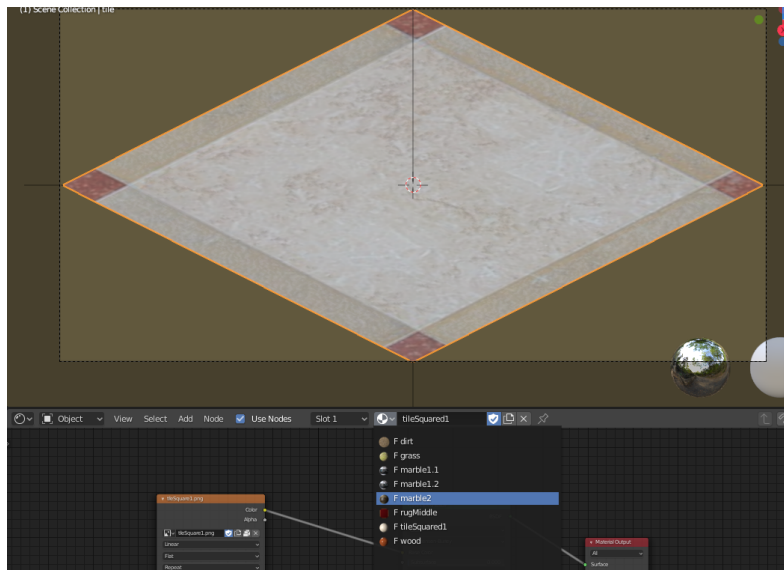


Figure B.10: A dropdown menu with available materials for a floor tile

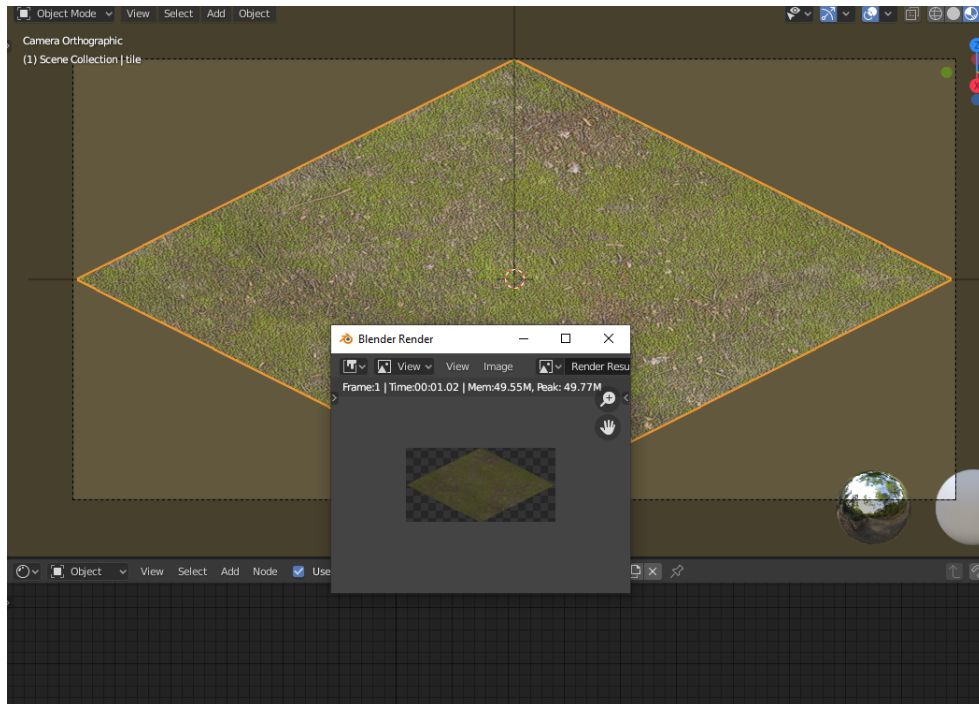


Figure B.11: Rendering a floor tile

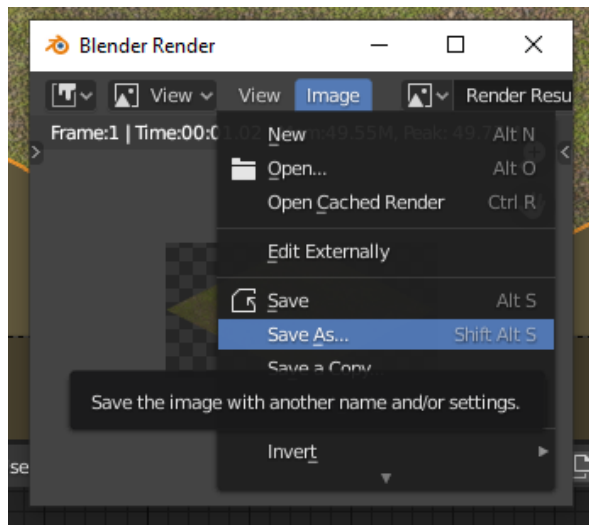


Figure B.12: Saving the rendered file as an image to a disc

B.5 Creating a sprite sheet and animation data using Spritesheet manager

This chapter explains how to use the Spritesheet manager application to create final sprite sheets and animation data.

B.5.1 Creating data for isometric characters

This chapter describes the process for the case when the input consists of 2D images, which contain some character, facing different directions. This is always output from the Blender rendering script described in this thesis. An example of such input is available in the `.\data\isometricCharacters\player walk\input` directory. This data can be used right away, when following this tutorial, if you do not want to render any 3D model to 2D images.

Run the application `SpriteSheetmanager1.0.2.exe`. Select all the rendered images from the Blender, with exception of the `originReference.png` file, and drag and drop these files onto the application, as depicted by figure B.13.

It is crucial that when you are drag and dropping the files, you drag and drop the first file in the sequence. The reason is that windows will pass the file you drag with the mouse as the first one to the application event. This causes that all the files will be sorted by name in the application, except the first one. The application might sort the files automatically, but there are cases when the user wants to define the order of the files by himself by dragging and dropping the files manually, one by one. That is the reason why automatic sorting was not implemented in the application.

B.5. Creating a sprite sheet and animation data using Spritesheet manager

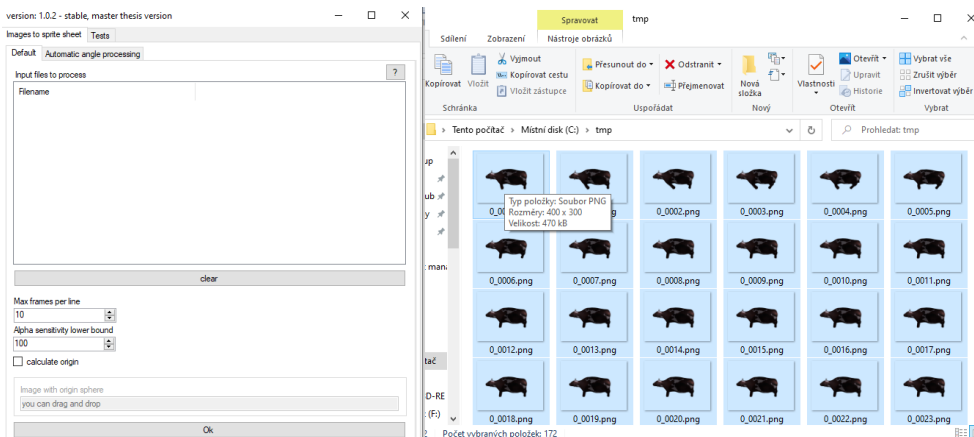


Figure B.13: Dragging the input images onto to the Spritesheet manager application

Tick the checkbox, which reads *Calculate origin*, and drag and drop the origin image containing the green circle, onto the textbox, as depicted by figure B.14.

The text in the textbox should change to the filename after drag and dropping it there successfully.

Go to the **Automatic angle processing** tab, as depicted by figure B.15, and tick the checkbox, which reads **enable**.

This means that the application will later create a sprite sheet for each rotation angle of the character and calculate the relevant animation data, as discussed in this thesis. If that checkbox would not be checked, then only one sprite sheet would be created from all the input data. The **Number of frames of animation** is the number of images per one rotation of the animation. As the input images are sequenced by numbers, have a look at the last image that was rendered to the final directory by Blender. Suppose that the last monster frame file has the name `7_0026.png`. Because the first frame is numbered from 0, then the number of frames of the animation is $26 + 1$, which is 27. Write the number into the discussed field in the Spritesheet manager. If the number is incorrect, the application will display an error.

The **Spritesheet name prefix** defines how the final sprite sheet file name will start. This is only for orientation purposes. It is advised to use a text which will describe the animation, such as **walk**, **attack**, **death**, etc.

The resulting final files will be created in the current directory, from where the Spritesheet manager is running, overwriting any already existing files. Suppose that prefix **run** was chosen in the previous step, then the following files will be created: `run.json`, `runD.png`, `runL.png`, `runR.png`, `runU.png`, `runLU.png`, `runLD.png`, `runRU.png`, `runRD.png`. If you are not sure, move the `SpriteSheetmanager1.0.2.exe` to a different directory, which is empty, and

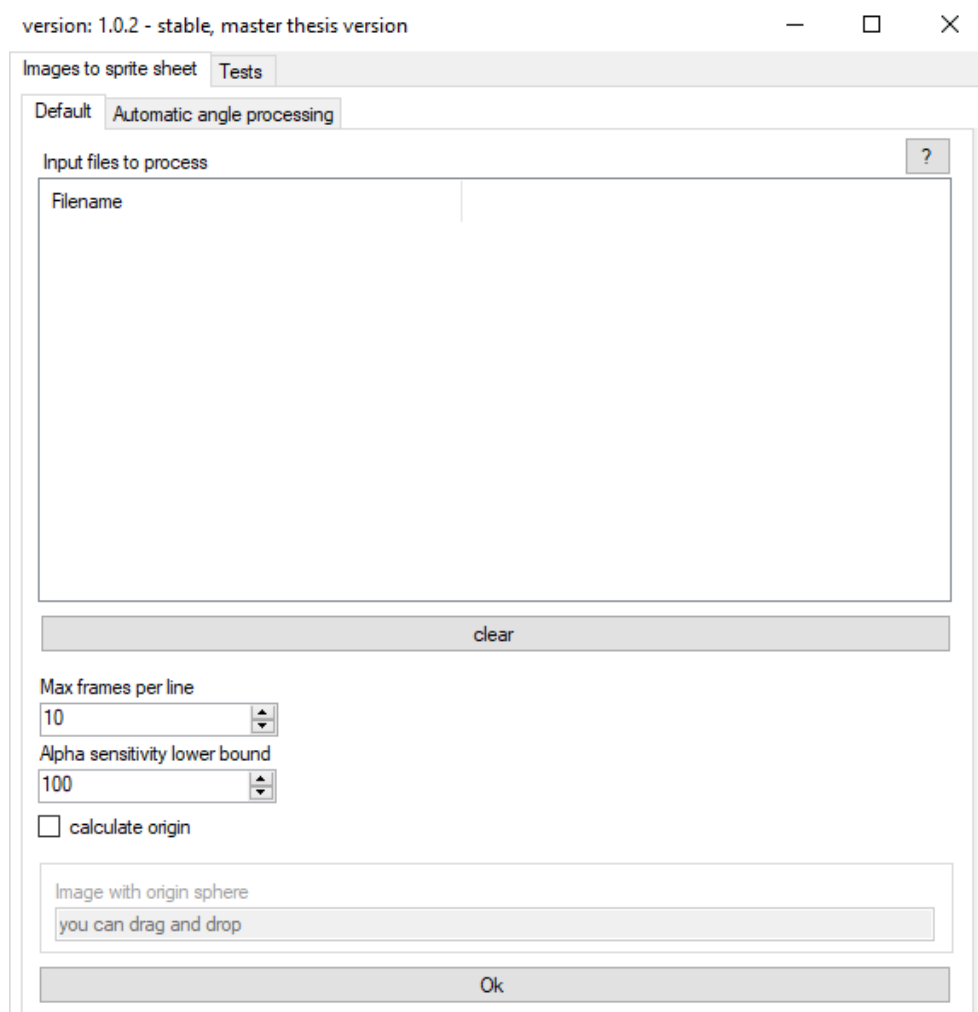


Figure B.14: Spritesheet manager application interface

run it from there.

If the input images contain semi-transparent pixels, change the **alpha sensitivity lower bound** value in the window. The importance of that value was described in the Alpha sensitivity lower bound chapter.

Go to the **Default** tab again, and hit the **Ok** button.

B.5.2 Creating data for any animation seen from only one angle

This chapter explains how to use the application to create an animation, which is not rotated to different angles. An example of such input can be seen in `.\data\frames to spritesheet\input`.

B.5. Creating a sprite sheet and animation data using Spritesheet manager

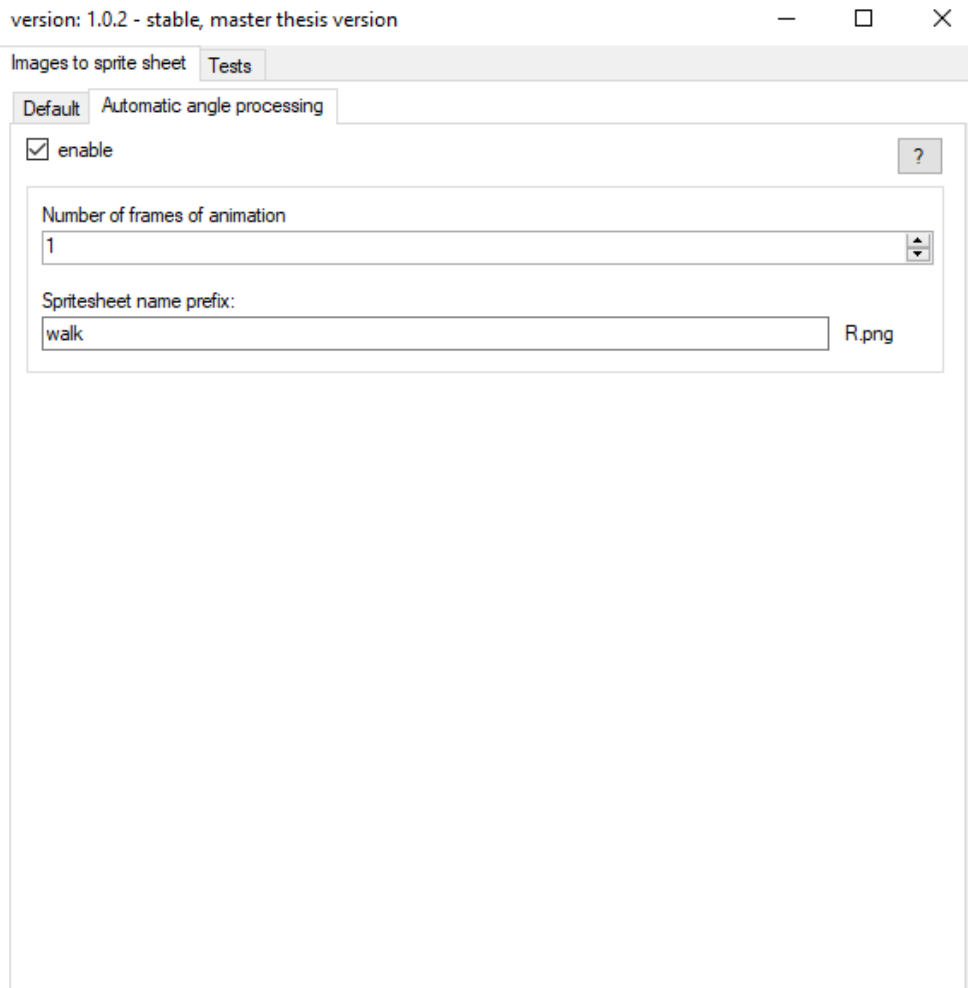


Figure B.15: The Automatic angle processing tab

Drag and drop all the input images, with the exception of the `originReference.png` file, onto the application, as displayed in figure B.13 (the figure displays different image data, but the idea of the drag and dropping process is the same). As described in the chapter `Creating data for isometric characters`, it is important to drag the first image with the mouse. If an origin point is calculated, you may provide the `originReference.png` file to the application in the same way as described in the chapter `Creating data for isometric characters`. In the directory described in this tutorial, there is no such file present, as it was not needed, so we will not check the `calculate origin` checkbox nor drag and drop the `originReference.png` file. As this is non-isometric animation, navigate to the tab `Automatic angle processing` and make sure that the `enable` checkbox is unticked. Then, navigate back to

the `default` tab and press the OK button. In this case, the result will be one `spriteSheet.png` (displayed in figure B.16) and one `spriteSheet.json` file.

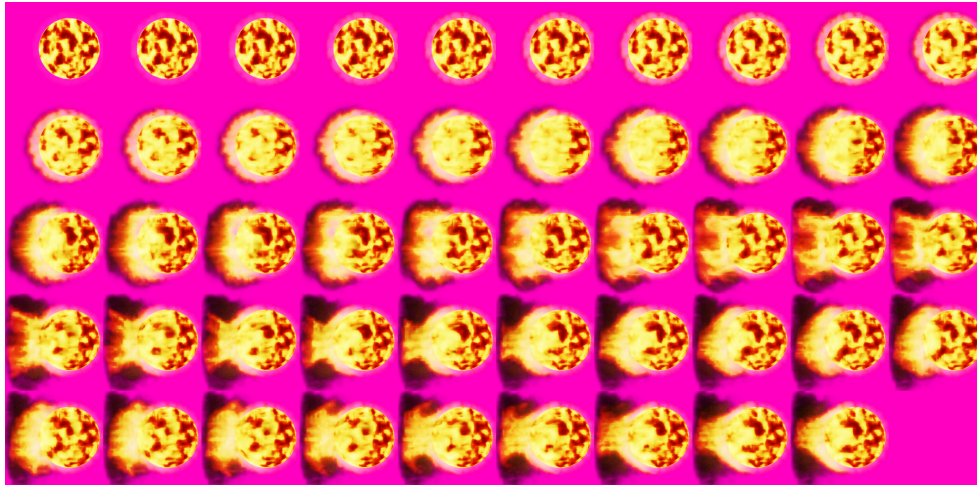


Figure B.16: Final sprite sheet for non-isometric animation. Transparent pixels are displayed as pink color in this illustration

B.5.3 Cutting off transparent pixels from one image

This chapter explains how to use Spritesheet manager to cut off unnecessary transparent pixels from a single image. Suppose that the input image is a couch, as displayed in figure 3.19.

This file can be found in `.\data\cutTransparent\input` directory. Run the Spritesheet manager application. Drag and drop the one image onto the application, in the same way as described in previous chapters. In this case, no origin reference point is needed, so `calculate origin` checkbox shall be unticked. Similarly, in the tab `Automatic angle processing`, checkbox `enable` shall be unticked. In tab `Default`, click the OK button. In this case, the result will be one `spriteSheet.png` (displayed in figure 3.20) and one `spriteSheet.json` file. As in this case, the `spriteSheet.json` file is not needed and can be deleted. However, if you choose to calculate the origin point, then the JSON file's data can be used later on.

B.5.4 Expected limitations

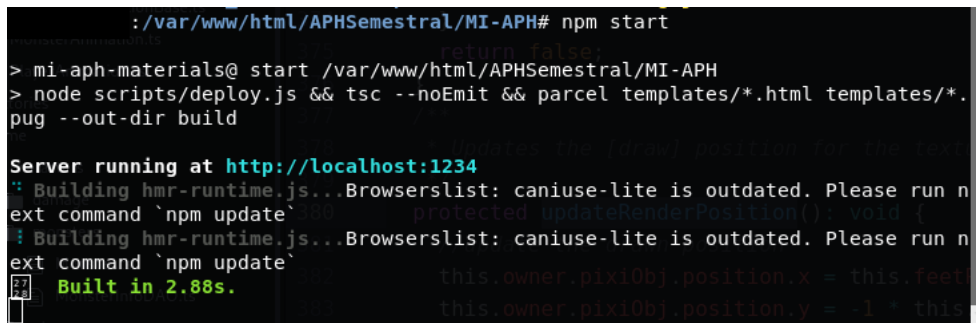
This chapter discusses the expected limitations for the Spritesheet manager application. The application's main objective is to help the user create sprite sheets and animation data, which would be a very difficult task if done manually. Considering input data, sometimes the origin point might be outside the minimal bounding box, calculated by the application. However, there can be

cases when this is the correct input. So the application processes it anyway. In short, the user needs to input logical data to get a logical result.

B.6 Running the game prototype

This chapter explains how to run the game prototype application. To run the application, a Linux operating system must be used. Concretely, the Linux distributions used were both **Ubuntu Desktop 20.04.2.0 LTS** and **Kali Linux**.

In order to run the game prototype, navigate to the `.\src\gamePrototype` directory in the provided sources, using the Linux terminal. Using a terminal, execute command `npm install`. This command downloads all necessary libraries that are needed to run the project. If you are unsure whether the installation is necessary, better run the command anyway. After the libraries are downloaded, run `npm start` command to run the server. The output can be seen in figure B.17. An alternative is to run the `compile.sh` script, which does the same as the `npm start` command.



```
:/var/www/html/APHSemestral/MI-APH# npm start
> mi-aph-materials@ start /var/www/html/APHSemestral/MI-APH
> node scripts/deploy.js && tsc --noEmit && parcel templates/*.html templates/*.pug --out-dir build

Server running at http://localhost:1234
: Building hmr-runtime.js...Browserslist: caniuse-lite is outdated. Please run next command `npm update`
: Building hmr-runtime.js...Browserslist: caniuse-lite is outdated. Please run next command `npm update`
Built in 2.88s.
```

Figure B.17: The terminal output when running the game prototype

Now, a web server will start on your computer, and the game will be accessible through a web browser. Navigate to the URL `127.0.0.1:1234` using your web browser, and you will be able to see the game, as seen in figure B.18.

B.7 Playing the game prototype

To move the main character, perform a left-click on the game area. The character will then move to the target place. The objective of the game is to collect all three brown bags, which are lying on the ground. When the character moves to the tile with a bag, the bag is collected automatically. A message about the collection event will appear in the game console, which can be seen on the screen's bottom-left corner. Upon collecting all three bags, a message window will pop up. If you click on the spider, the character will go

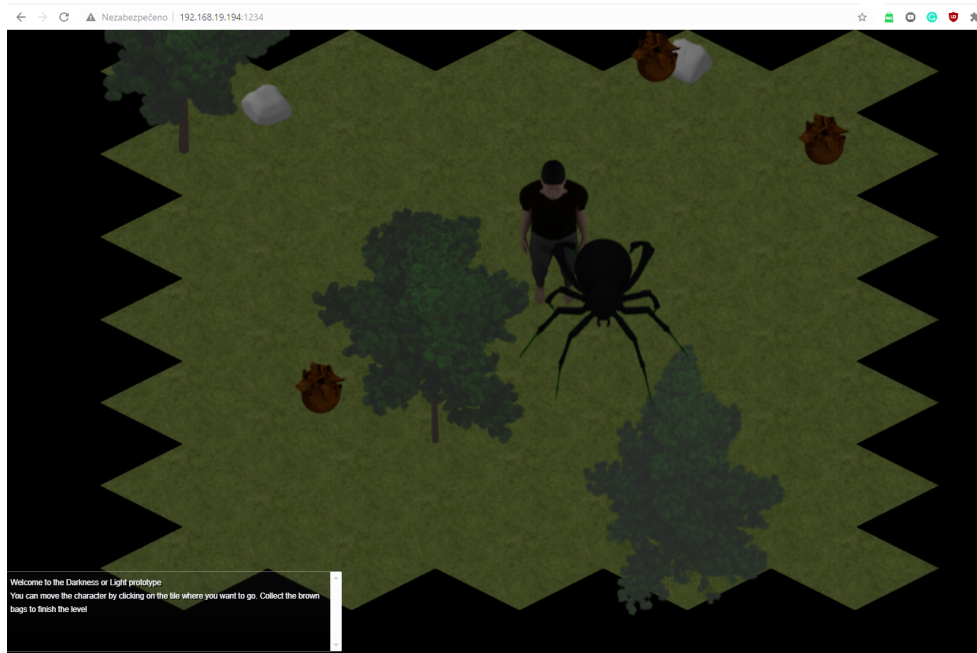


Figure B.18: The game prototype, when navigating to the localhost in a web browser

and attack the spider. The spider, however, can not die, so the fight will last until you click on the world again to navigate the character somewhere else. This feature displays the character's attack animation and a change between the walk animation and attack animation.

B.8 Running the tests

This chapter describes how to run the tests.

B.8.1 Blender rendering script

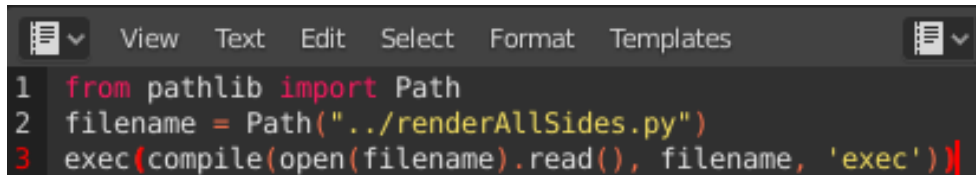
This chapter explains how to run the tests for the Blender rendering script. Here is described how the script which allows the automatic rendering from Blender was tested and how to run the tests.

Automated tests

This chapter explains the automated tests work, and how to run them. Blender itself does not provide any features that would allow testing a script code. Traditionally, a custom script is written to test some correct implemented functionality. Having the script `renderAllSides.py` that we want to test, one

possible way to implement the test would be to add the tests to the script, and based on a command-line argument, and a test would be run.

This would, however, require running the script from the command line and not from the Blender interface, as I have not found a way to pass a command-line argument to an `exec` function, when the script is compiled and run from the GUI, as depicted by figure B.19.



```

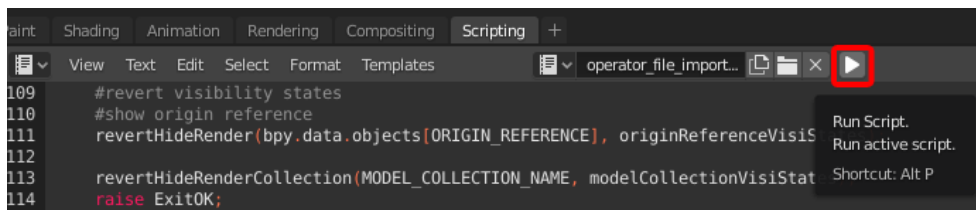
1 from pathlib import Path
2 filename = Path("../renderAllSides.py")
3 exec(compile(open(filename).read(), filename, 'exec'))

```

Figure B.19: Using `exec` function to compile a script and run it

The other way, which is the way I chose, was to create a new script, which would implement only the tests. Such a script is called `tests.py`, and contains a copy of functions from the `renderAllSides.py` script. The tester tests these functions. The reason for such a solution is that there is no way to include the functions from the other script. It would be possible to use the `compile()` function, which returns the specified source as a code object, which can be executed. Such an execution would allow to run the entire script and not only various selected functions.

For running the tests, Blender must be installed. Then open the file `.\bin\blender\test\test.blend`, which can be found in the sources. Then navigate to the `scripting` tab. From here, press the button with an arrow to run the script, as depicted in figure B.20.



```

109 #revert visibility states
110 #show origin reference
111 revertHideRender(bpy.data.objects[ORIGIN_REFERENCE], originReferenceVisiS
112
113 revertHideRenderCollection(MODEL_COLLECTION_NAME, modelCollectionVisiStat
114 raise ExitOK;

```

Figure B.20: Running a script in Blender

For showing the Blender's console, click on the `Window` tab and select the `Toggle System Console` option, as shown in figure B.21. A console window should appear. If it does not show up, choose that option again, and the window will show up. The reason is that probably, for the first time, the console window is shown but is hidden behind the Blender window. The second command brings the console window to the front.

The test result will be printed to the console. An example output can be seen in figure 4.1.

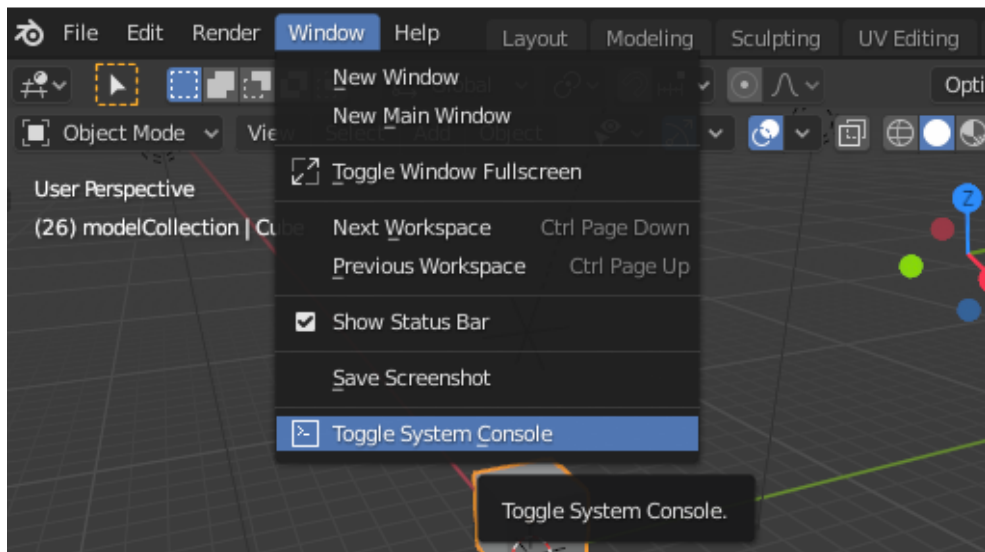


Figure B.21: Showing a console window in Blender

Manual tests

This chapter explains how manual tests can be executed. This process requires you to use the rendering script to render the model. Then there are already provided reference data so that you can compare whether the render output matches the expected one. To do so, navigate to the directory `.\bin\blender\playerModel` in the provided sources. In such a directory is a directory `output_walk`, which contains the expected output. Render the model, which can be found in file `.\bin\blender\playerModel\player1_0`, and render the `walk` animation. The process of rendering is explained in the chapter Using an existing 3D model in Blender and exporting to 2D images. After the process is done, the resulting data shall match the data which can be found in the mentioned `output_walk` directory.

B.8.2 Tests for the Spritesheet manager

To run the Spritesheet manager's tests, you need to run the application `.\bin\SpriteSheetManager1.0.2.exe`, which can be found in the sources. For running the tests, you need to select the `tests` tab, as displayed in figure B.22.

After that, upon clicking on the button, a tester window will open, as displayed in figure B.23. The tester features are not directly in the tab to separate the tester user interface from other user interfaces for easier usage and clarity. After clicking on the button which reads `Run test`, automated tests will run, and the result will be printed to the window.

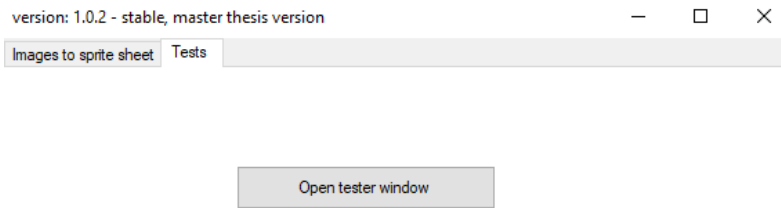


Figure B.22: Tests tab

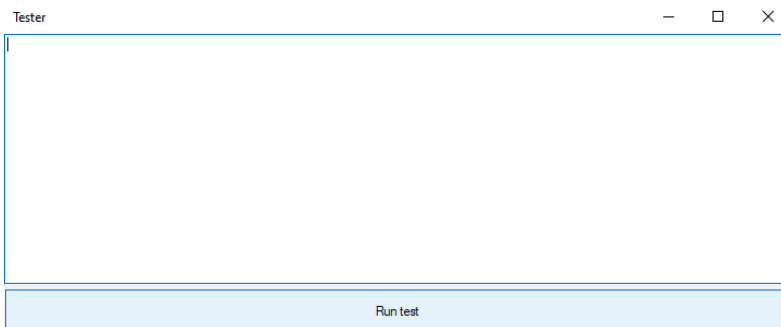


Figure B.23: Tester interface

In figure 4.2 can be seen an output of passing automated tests.

Content of the data media, provided along with this thesis

For more detailed description of the data media content provided along with this thesis, please read `README.md`, which can be found on such a media. Also, please note that the content of the provided data media and the GitLab repository [22, 1] is the same.

C. CONTENT OF THE DATA MEDIA, PROVIDED ALONG WITH THIS THESIS

README.md.....	Description of the repository
data.....	Contains input data for the Spritesheet manager application
frames to spritesheet	Data for when classic frames shall be converted to a sprite sheet. Does not involve isometric characters
input	The input data
output*.....	The expected output data
isometricCharacters ..	Data for when frames of a isometric haracter shall be converted to multiple sprite sheets (for all 8 angles)
player walk/input.....	The input data
player walk/output	The expected output data
floortile	Data related to the floor tile creation
floortiles.blend	Blender file containing a scene from which floor tiles can be rendered
output/*.png.....	The resulting floortile textures
cutTransparent	Data which can be used to cut off unnecessary transparent pixels from the image, using the Spritesheet manager
input/*.....	The input files
output/*.....	The expected output from the Spritesheet manager
bin	Contains executable form of the Spritesheet manager, and also blender files, which can be used for exporting the 3D model to a 2D images
blender	
playerModel	Model of a player
output_walk.	The expected output, after rendering the player's model's walk animation, using the rendering script. This serves as a reference for manual testing
template	A file which can be copy pasted to create any new model, and keep the scene setup
test	Contains blender file which can run the tests
renderAllSides.py	The rendering script, which is referred to from the blender files
SpriteSheetManager1.0.2.exe.....	The binary of the Spritesheet manager application. The newest version
src	Contains the implementation source codes
gamePrototype	Source code for the game prototype. This also can be run (as the game prototype is a small game server, which will run locally)
spriteSheetManager.....	Source codes for the Spritesheet manager application