Game Development

Using open source software

Emil Melander

Examensarbete / Degree Thesis
Informationsteknik / Information Technology
2020

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	7690
Författare:	Emil Melander
Arbetets namn:	Spelutveckling med programvara baserad på öppen källkod
Handledare (Arcada):	Dennis Biström
Uppdragsgivare:	

Sammandrag:

Många små spelbolag och oberoende spelutvecklare har en strikt budget när de vill skapa någonting nytt och de har inte alltid råd med dyra licenser för programvara menat för utveckling. Syftet med detta examensarbete är att beskriva och analysera utvecklingsprocessen av ett datorspel med att endast använda gratis programvara som är baserad på öppen källkod. Problemen som utforskas är om läsaren kan använda sig av liknande metoder och programvara som har använts i detta examensarbete och hur arbetsflödet mellan programvaran lämpar sig för utveckling. Examensarbetet börjar med att förklara målen, bakgrunden och det allmänna spelupplägget som det praktiska delen av arbetet består av. Arbetet fortsätter med att klargöra hur de tredimensionella modellerna skapas och animeras i Blender men även förklarar allmän teori om tredimensionella modeller som kan tillämpas i andra projekt. Texten går igenom programvaran GIMP och hur det kan användas för att skapa texturer och effekter för datorspel. Eftersom spel är audiovisuella erfarenheter, fortsätter den teoretiska delen att beskriva hur ljud och musik skapas i Audacity och LMMS. Slutligen förklaras hur allting knyts ihop i Godot spelmotorn med C# kod och verktyg som är tillgängliga. Slutresultatet är ett fullt funktionerande datorspel vars spelupplägg är intressant och roligt.

Nyckelord:	Spelutveckling, Öppen källkod, 3D, C#
Sidantal:	39
Språk:	Engelska
Datum för godkännande:	

DEGREE THESIS
Arcada

Degree Programme:	Information Technology
Identification number:	7690
Author:	Emil Melander
Title:	Game development using open source software
Supervisor (Arcada):	Dennis Biström
Commissioned by:	

Abstract:

A lot of small game studios and independent game developers have a tight budget when they are creating something new and necessarily cannot afford expensive software licenses for development. The general purpose of this thesis is to describe and analyze the development process of a video game by only using free open source software. The problems that are researched in this thesis are the following: can the reader use the very same techniques and tools that are being used in this thesis and how viable is the workflow between the software for development. The thesis starts by explaining the goals, background and the general gameplay elements that the practical part of this thesis contains. It then proceeds by explaining how the three-dimensional models were constructed and animated in Blender but also some basic model making theory that can be applied in any project. It also talks about GIMP and how that particular software can be used to create textures and effects. Since games are audiovisual experiences, the thesis continues by describing the audio design workflow in Audacity and LMMS softwares. Finally, the thesis wraps up everything by explaining how everything is tied together in the Godot game engine using both C# code and the tools and classes that are available in the game engine. The final result is a fully-fledged completely playable and entertaining video game.

Keywords:	Game development, Open source, 3D, C#
Number of pages:	39
Language:	English
Date of acceptance:	

CONTENTS

FIGURES

1 INTRODUCTION

1.1 Background

What separates games from other entertainment and media, are the challenges and reward systems that are present in most of them. The player is presented with a challenge that they must solve using a set of inputs. After the player has completed the challenge, they are subconsciously expecting some sort of reward. This reward can be as simple as a sound that confirms that the player has completed an action or a number on the screen that keeps growing throughout the gameplay. (Wang & Sun 2011) Developing a game as an independent developer, can be a daunting task. There are multiple different aspects the developer has to take in consideration when making a decision about the game, whether it is gameplay, visuals or the sounds. In this day and age, there are a lot of different tools and software that help you to create games and speed up the workflow. Most of the commercial game engines and software have some kind of restriction or license that you have to follow or pay for, in order to release your game to the general public. (Xinogalos, 2017) This is where the world of open source comes in play. As a counterweight to the commercial closed source software, there are plenty of extraordinary open source projects that are equally good, if not better. They are made and maintained by huge communities and are driven by passion in general. (GitHub, 2020) This means that open source software have a lot of features that the community wants as they get implemented fairly fast compared to a company that has to communicate with the users back and forth. There are not a lot of games that have been with made with open source engines if you look at the data from 2019 and that is why this thesis aims to test out how applicable the workflow is using one. (Toftedahl, 2019) Instead of comparing open source software and solutions with commercial closed source software and solutions, this thesis will focus on showing the reader the workflow of creating a functioning game with open source tools. It goes through the different software and their user interfaces and features and then proceeds to show the reader how they were used in the practical part of this thesis. This way the reader should get a good overview of how games are made and how viable open source is for video game development.

1.2 Purpose and goals

The theoretical written part of this thesis will walk through each step of the development process and will explain the different aspects of each software in detail and reflect both the positives and negatives of the whole project workflow and give the reader an overview of what it is like to work with the listed tools and how games are made when you are an independent developer. The practical part of this thesis is to produce and develop a perfectly functional and playable video game by only using open source software listed. It is important to create compelling interactive gameplay with the help of various algorithms, logic and gameplay theory. This is implemented by programming a system that generates complex game levels with variety and interesting gameplay mechanics regarding the challenges in the game. The game should be able to render three-dimensional worlds and graphics flawlessly. This includes models with textures, materials and animations. These combined with shaders and a user interface should create the visuals for the practical part. The game also needs to have sound effects and music, which are made with the help of sound theory. The sound effects will be recorded with personal hardware, while the music will be made with ready made samples that are included in the free open source software. All the assets will be bundled and tied together in a game engine with C# as a programming language and with the help of visual tools to create a fully-fledged game. The whole purpose of this thesis is to test how feasible open source tools are for game development and how smooth the workflow and transitions are between software.

1.3 Limitations

Since video games can be anything and time is limited, they cannot be everything. The project size had to be confined, so that the core gameplay events were present. One could have added a lot of things that could have enhanced the experience since the gameplay flow is very modular. For example: powerups, a bigger variety of enemies or treasures hidden in the dungeons. This thesis will not explain the basic theories and practices of three-dimensional modeling, graphics design or audio design.

1.4 Tools and software

Godot	Godot is an open source game engine that has a wide variety of tools and settings. It supports both two-dimensional and three-dimensional elements that you can mix however you like. You can write logic in their own GDScript or C#. It is possible to write in C and C++ as well.
Blender	Blender is a three-dimensional modeling and animation software. You can export models to multiple formats with UV maps and animations.
GIMP	GIMP or “Gnu Image Manipulation Program” is a piece of software that, as the title suggests, helps you to manipulate or create two dimensional images.
Audacity	Audacity is an audio recording and editing software.
LMMS	LMMS or “Let’s Make Music” is a free, cross-platform music making software with a huge audio library.
C#	C# is an object-oriented programming language originally designed by Microsoft. I am using the Mono compiler, which is an open source alternative bytecode compiler.

2 GAMEPLAY

2.1 General

The practical part of this thesis itself is an infinite dungeon crawler video game. You play as a knight with a crossbow and your goal is to find the stairs on each floor of the dungeon, while dodging the attacks of the imps that live there. There are two kinds of imps: melee and ranger. The melee imps run up to you and they can only attack when they are close and the ranger ones try to get to a certain distance and then they fire a spell towards you, which you have to dodge. At the beginning of the game, the player has three hearts when they start. If an imp manages to hurt the player with an attack, the player loses a heart. The game ends when you run out of hearts and you see a screen that shows you how many imps you have slain and how many levels you have progressed. The imps have a similar system as the player to check whether they are alive or not. Different types of imps have different amount of hearts as well as different damage factors. This decision was made, so that one could potentially in the future change the difficulty and make changes to the gameplay. The gameplay elements are straight forward in this specific video game. The player is presented with a challenge: dodge the attacks, optionally attack back and find the stairs. The psychological rewards are the increasing numbers of slain imps and levels progressed and the sounds that the imps make when they are damaged or defeated, which all count towards satisfaction, which is an important aspect of a video game to keep the player hooked. (Wang & Sun 2011) The goal with the procedurally generated dungeons and content is to keep the game interesting for a longer period of time. In theory the system should feed the curiosity of a player longer than having the same dungeon floor with the enemies in the same place every time. All of the elements described above should contribute to keeping the player active and interested in the game, while also should not be over complex to implement in practice.

Figure 1 below shows “Ritari” gameplay and the described elements of the game: two different imp types, procedurally generated three-dimensional dungeon and the goal stairs.

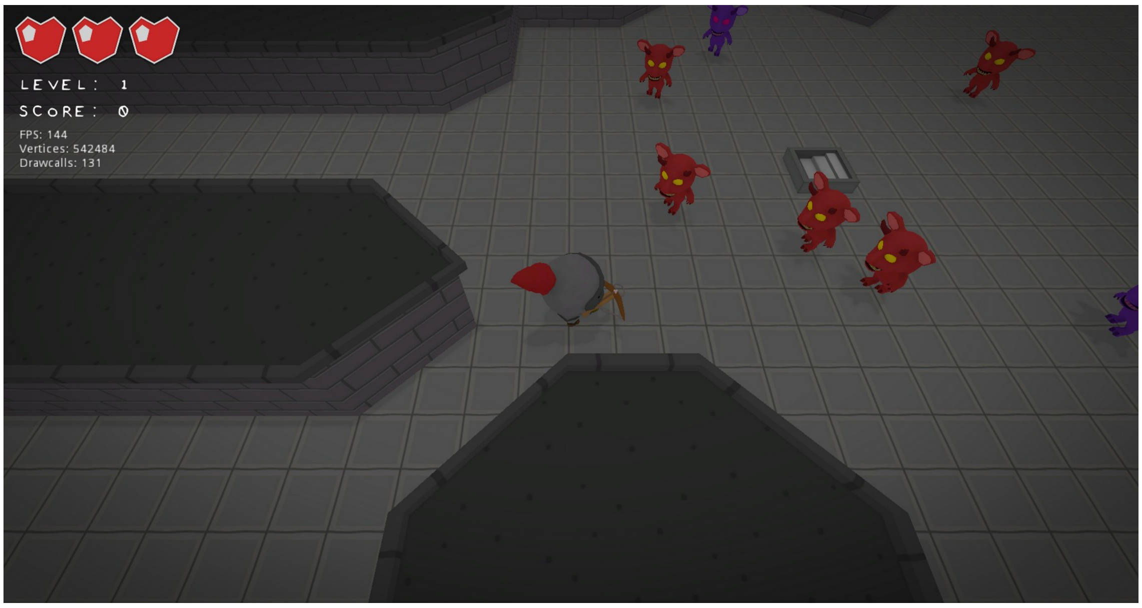


Figure 1. "Ritari" gameplay

2.2 Algorithm

The dungeons are procedurally generated with an algorithm that uses cells to build rooms and corridors. One could try other algorithms like cellular automaton and pseudorandom noise functions to produce more organic looking dungeons. The algorithm produced in this thesis, generates different sized rooms and corridors of different length to make it feel like it was made by something living for the player. The dungeon itself is built in a cellular way. The algorithm starts by initializing two separate two dimensional arrays. The first array contains integers, that is being used to describe whether a cell contains a wall or not by simply storing a one or a zero at a specific index. The second array uses the same technique but it is used for the game entities instead, which is in this case the imps. One could also store other interesting elements like powerups, traps or loot. After both arrays are initialized, the algorithm fills the wall array with walls, because it is going to carve out the dungeon in the next steps. In order to get the rooms, the algorithm creates another two dimensional array, which is a significantly smaller and only contains information about the rooms. The size of the array equals to the divider constant multiplied with itself, which the algorithm uses to divide the main wall array with and to decide what size the rooms should be. The next step is to generate random rooms in that array. The code randomly searches for empty slots in the array where it can put a room, until it has reached the amount of rooms wanted. It then proceeds to loop the smaller array and checks if a slot contains a room or not. If it contains a room it scales the position to the wall array and with a nested for-loop carves out the room. All the rooms are the same size. However, if two rooms get placed next to each other, they will form a bigger room. The rooms also need to be connected together. The position data of the rooms is stored in a list, which can be iterated through. The dungeons get connected in the same order they were placed, meaning, that you will always get a dungeon without isolated rooms. This way you will also get really complex looking dungeons with a lot of variety. The enemies are added by looping through the empty dungeon and checking if a cell contains a wall or not and if a certain probability is met. Finally, the code adds the player in a randomly selected room, clears the enemies nearby so that the player will not die immediately when they start the game and adds the stairs in another random room. Figure 2 shows six different dungeons generated by the algorithm that is present in the game and described above. The green dot represents the player spawn point, the red dot represents the stairs and the blue dots represent enemies.

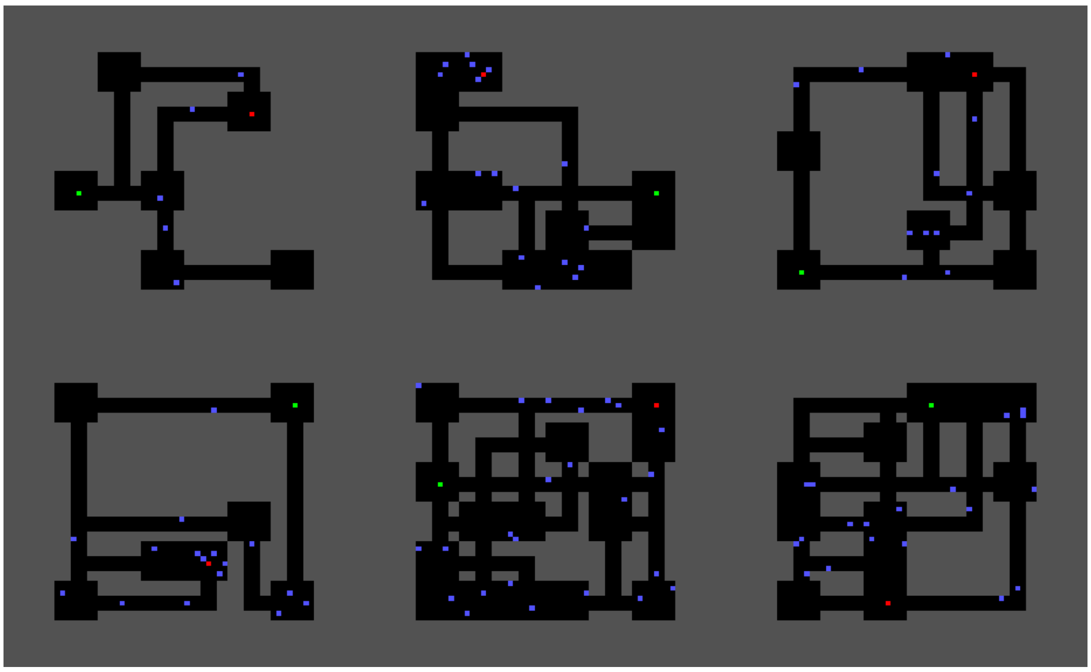


Figure 2. Procedurally generated dungeons

3 BLENDER

3.1 Blender 2.8 user interface

Blender has changed its user interface drastically from version 2.79. The new interface that was introduced in version 2.8 was meant to be more intuitive and responsive compared to the old one. The top bar has different tabs for different purposes for the model making process, which helps the end user to keep things organized and clean. These tabs are the generic layout, modeling, sculpting, uv editing, texture paint, shading, animation, rendering, compositing and scripting. On the left side you have tools that you might need inside the tab you currently find yourself in. For example if you are inside the sculpting tab, you find your sculpting tools on the left side. On the right side you have the general options for your scene, model, render settings and the outliner hierarchy tree. The center view changes depending on what one is working on at a specific moment, it could be the three-dimensional mesh model or the uv maps and textures for example. Blender is also very modular, the user can create a custom tab and define where all the subwindows are positioned or even have them outside the main frame of Blender on another monitor for example.

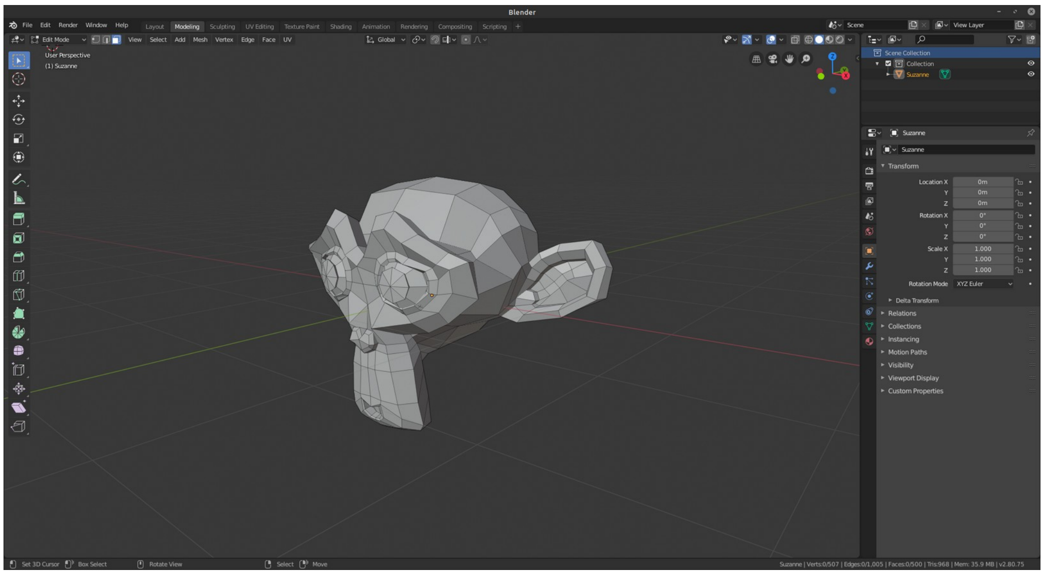


Figure 3. Blender 2.8 user interface

3.2 Blender Workflow

3.2.1 Modeling

The workflow to create models and animate them for a video game with Blender is straight forward. It is advised to start with a low polygon primitive shape like a cube or sphere and then expanded from there. It is not considered a good practice to start with a high polygon model and sculpt the model from there, because the polygon count would be too much to handle in real time inside the video game and it is a lot more work to remodel the model with a new lower polygon count topology. The building blocks of the practical part are the tiles that make up the dungeons. They are low polygon models consisting mostly of quads with a cartoony texture applied to them. To make the models fit together when they are side by side, they have to have some vertices that share the same position, share the same size and have a seamless textures uv mapped correctly. This way when the tiles are exported to a game engine, it is easier to tweak their size and position by code and implement them into a larger system. The origin of the model is important to set right as well in Blender, using a model with an origin that is offset by a lot can cause problems when you are trying to position it properly.

Figure 4 below shows how two tile models fit together with the origin set to the floor.

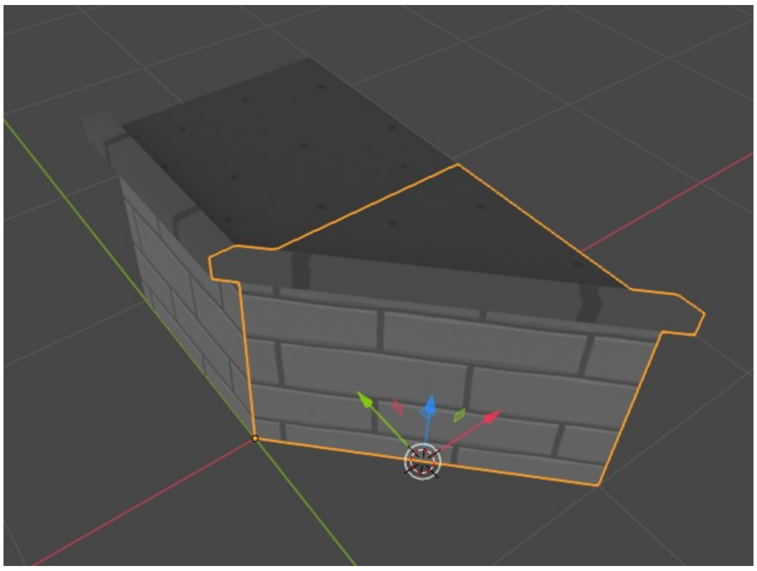


Figure 4. Two tile models

When modeling a character, the goal is to create a model where the model stands in a

“T-pose”, a pose where the character stands with the arms wide open as possible in the shape of the letter t. This is to make it easier to build a skeleton and see the symmetry. In order to make the model of a character symmetrical, the mesh has to be split in half and the other half that is not symmetrical has to be removed. Then by using a mirror tool, the mesh will become flawlessly symmetric. One has to keep the topology in mind when modeling a character, for example if a character is supposed to make a lot of facial expressions, one might consider adding more polygons around the face and less around the other body parts. In order to get a cartoony look, you can tweak the character proportions by scaling different parts of the mesh differently. Details like ears, horns and teeth, can be made by selecting a specific area on the mesh and extruding it.It is important to remember the vertex count when designing three-dimensional models for a video game. One has to consider the hardware the game is targeted towards. For instance, the difference between mobile and desktop hardware is substantial. The more models you have to render on the screen, the more vertices and taxing it becomes for the hardware to calculate. (Godot Engine, 2020) It is a good practice to remove unnecessary polygons that are not visible or do not contribute to the overall look of the model in a meaningful way, merging multiple faces and creating a clean topology that consists of quads. When you are working with a mesh, you might accidentally create inverted normals. Normals are vectors that define what direction a polygon should face towards. This is fixed by selecting all the faces of the model and using Blender’s internal tool to recalculate the normals of all the faces. Figure 5 shows two models: on the left is an unoptimized imp model, on the right is an optimized version of it. The optimized model might look crude when you look at it this close. However, it is really hard to see a difference in game when the models are far away from the camera. Optionally, one can generate a normal map from the high polygon model and apply it on the low polygon model to smooth out the hard edges.

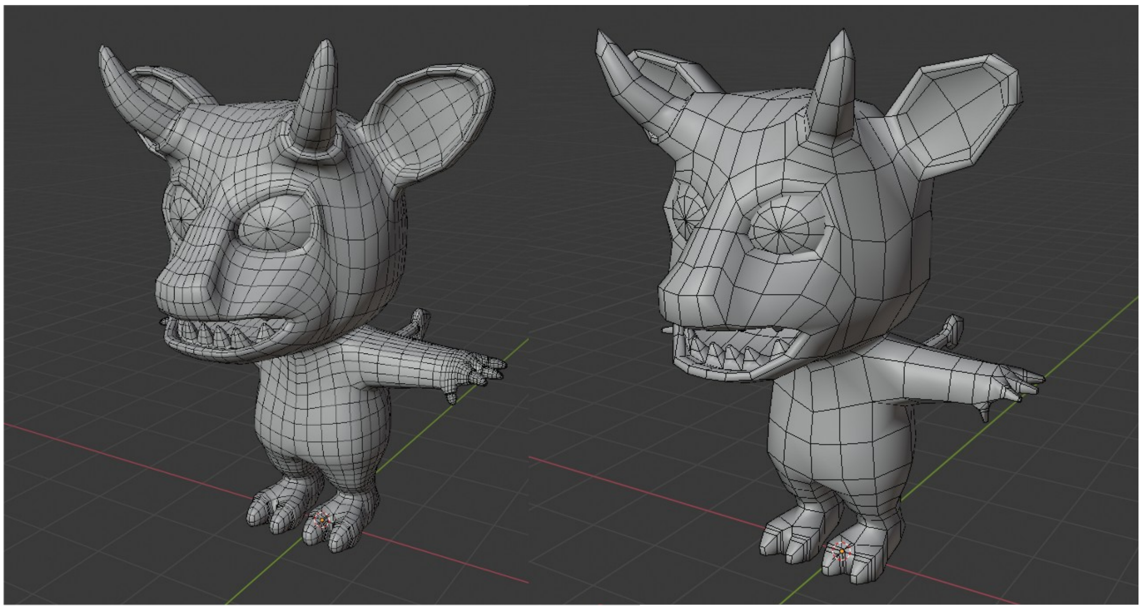


Figure 5. Model optimization.

3.2.2 Texturing

In order to apply details and textures on models, they have to be UV mapped. UV maps represent a surface on the model. Imagine you have a standard cube with six sides. When you unfold the cube, you have all the sides as quads as a two-dimensional shape. If you draw something on one of the quads, the drawing appears on one of the faces of the cube. Blender has multiple different tools that are focused on UV mapping. One of them unwraps the model’s faces automatically and places the unwrapped faces in way they will not collide with each other. When a model is UV mapped, you can start painting textures inside Blender using the internal paint tools or you can export the UV map layout as an image, that you can paint over in another software. The Blender paint tools support different brushes and patterns. One can also adjust the smoothness curves and opacity of the paint tool. Since real time screen space ambient occlusion is a very taxing operation to execute, it is advised to bake ambient occlusion maps with the texture, especially if you have a lot of visible objects on the screen. Blender has a tool that generates an ambient occlusion map that you can either combine inside Blender or export it as an image. To adjust the graininess of the ambient occlusion map inside the cycles render engine, one can adjust the sample count. Figure 6 shows an example of an automatically generated uv map of the imp model and ambient occlusion. The uv map is far from perfect and it would be really difficult to paint any details on but works alright when you only have colored areas as your only detail.

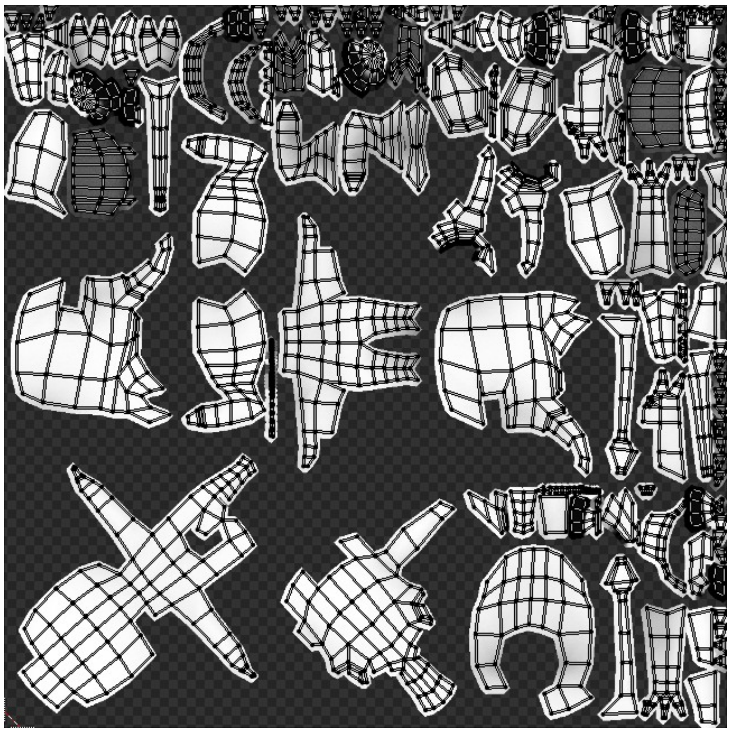


Figure 6. Automatically generated uv map

3.2.3 Animating

To animate a model you need to create a skeleton, which consists of bones that affect the model’s mesh vertices. Blender has a function to automatically calculate the weights of the bones. The weights affect the vertices around the bone, meaning that if you move the bone, the vertices affected move too. The automatic calculation works most of the time flawlessly but if the model has a lot details nearby, it might cause issues where the details around the bone that should not be affected, get affected. These problems can be fixed by manually telling Blender what faces of the model it should morph when you rotate the bone.

Blender also allows you to move the bones with automatic inverse kinematics, which means that when you move a bone it moves the parent bones in the bone hierarchy too. This enables artists to create realistic natural movements and leaves no bone without motion. To animate the model you work with something called keyframes. A keyframe is just a point in the timeline that tells the computer how the bone or object is oriented and where it is. By having multiple of keyframes, the computer can linearly interpolate between the transforms and create a smooth looking animation. Depending on the format the model gets exported to, it saves the keyframes to the file, which then can be used by a game engine to play animations. Figure 7 shows the right shoulder bone affecting the vertices around the shoulder. The color shows how strongly the vertices are affected linearly from blue to green to red. The character in the picture is also standing in the famous “T-pose”. Figure 8 shows the animation workflow components. On the right you can see all the bones of the skeleton in the green list. The white dots right of the list represent the keyframes. On the left you can see how the keyframes together with the weighted bones affect the model mesh.

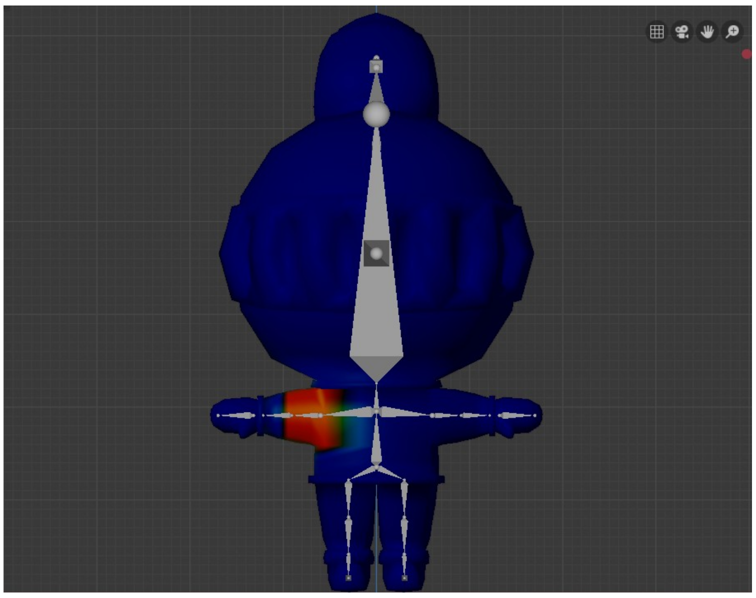


Figure 7. Model armature weight.

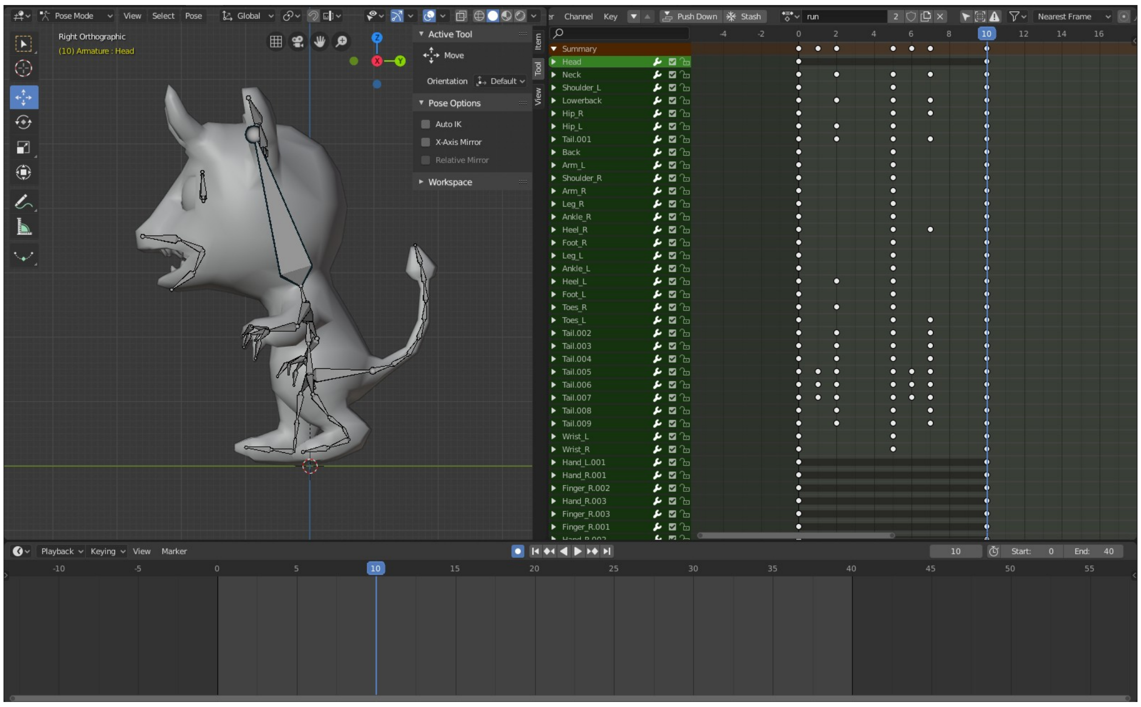


Figure 8. Blender animation workflow

4 GIMP

4.1 GIMP user interface

GIMP has a very clean and organized user interface, on the left you have the toolbar that contains all the tools you can manipulate the pixels on your canvas with. On the right you have the layer system, where you can manage the different layers of your image. You also have a wide variety of different brushes, textures and fonts in a selection area to make it more organized. At the top you have different tools and settings that help you to create different desired effects. GIMP is also highly modular when it comes to modifying the user interface. You can drag and drop the different toolbars and dialogs in different places and even have them outside the main software frame. This gives the user a lot of different options to customize the workflow and feel.

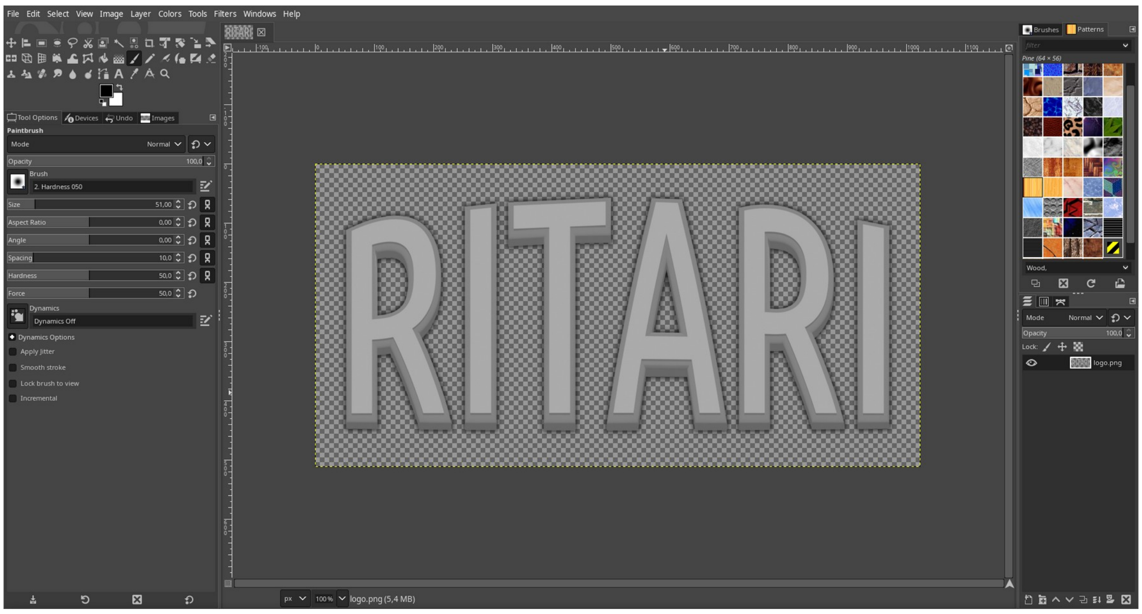


Figure 9. GIMP 2.10 user interface.

4.2 Textures, effects and user interface

The visual appearance of the game dungeon is built by using a set of three-dimensional tiles with different textures. These tiles need to be seamless in order to blend in with each other. Inside GIMP you can toggle an option that allows you to paint symmetric textures in three different styles: mirror, tiling and mandala. For the practical part of this thesis the tiling tool was chosen, as it makes sure that the pixels align with each other both on the x and the y axis, giving us a seamless effect. The game also contains a lot of different bitmap effects that are used in shaders. By using the noise generator tool, you can create a seamless noise that is useful for transition shaders and effects. To give the game a more grunge feel, one can create a vignette image in GIMP. To create one you cut out a circular shape from a solid black image and then feather the edges heavily. The interface in the game is minimalistic and effective in order to avoid clutter that would cause confusion. The main menu background was created by taking the wall texture and tiling it a few rows and columns. Applying a blur filter makes it less distracting when you look at it. The logo and the buttons that you can see in the main menu are three-dimensional models that have been rendered in Blender. The basic render settings works fine but sometimes you need to fix the contrast and saturation in GIMP. To make the interface pop out from the background, you can add a shadow by creating an outline behind the rendered image, adding a blur filter and decreasing the opacity a little bit. One of the more daunting tasks when it comes to video game user interface, is to create a font for the game. One could use a ready made dynamic font but to get a truly unique feel, a custom bitmap font should be considered. The game has a custom bitmap font, where each character is 64 by 64 pixels in size and when you take the generic english alphabet, the numbers and a few special characters, you get an image size of 1664 by 128 pixels.

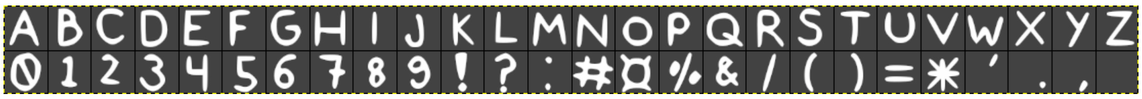


Figure 10. Bitmap font of "Ritari"

5 AUDACITY

5.1 Audacity user interface

The interface of Audacity is fairly simple and straightforward. You have your buttons for playback and recording at the top with bright colors and symbols. Next to them you various tools that you can manipulate your audio tracks with. The tools include cutting, copying and pasting. In the middle you have your audio tracks that you can manipulate. The audio tracks have sliders for volume gain and balance, as well as options for different formats and toggling between mono and stereo. On top the playback buttons you have different drop-down menus for different purposes. Few important ones are the generate, effects and tools menus. The generate menu contains tools that allow you to generate different sounds with different algorithms and formulas. For example you can generate a chirp sound by using a sine wave with a specified start and end frequency, amplitude and interpolation. You can then edit the sound by applying one of the many available effects from the effects drop-down menu. The tools menu contains settings for setting up macros, that help you to automate effects. It also contains tools for exporting and importing sample data.

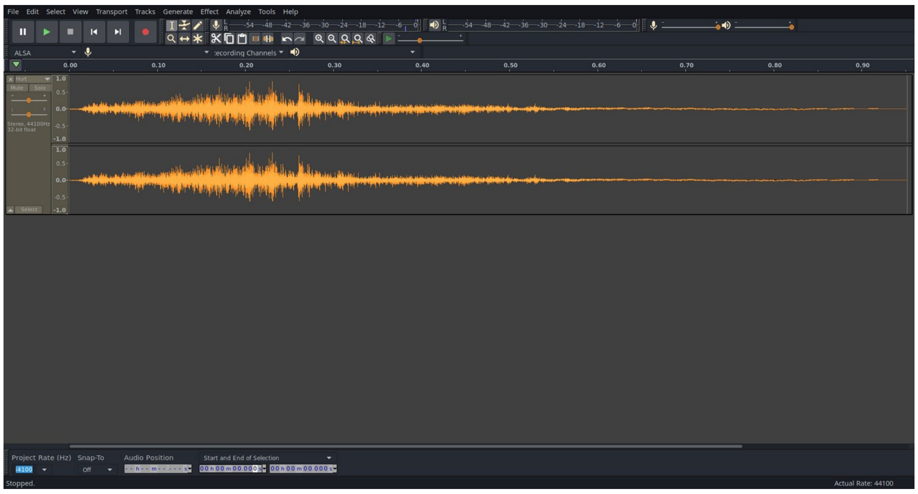

File Edit Select View Transport Tracks Generate Effect Analyze Tools Help
Project Rate (Hz)
Snap-To
Audio Position
Start and End of Selection
Stopped.
Actual Rate: 44100

Figure 11. Audacity 2.3.3 user interface

5.2 Recording and editing the sound effects

A game can be good without sounds but to add a whole new dimension and depth, a game needs sounds. As described earlier, sounds work as psychological rewards and indicators for gameplay elements. The sounds for the game were recorded using a condenser microphone in order to get good quality sound samples. The imps sounds were created by slowly cutting scissors nearby the microphone. The sound sample was then stretched out by slowing the tempo, resulting in a very eerie snarl sound. By turning up the pitch, the sound would fit the cartoony look of the imps better. The result was a demonic screech that was not too scary but not too kind either. The crossbow shoot sound was made by mixing a thump, a woosh sound made by a wooden stick swinging in the air and a ruler rumbling on the table. In order to get the sound feel right, the bass of the thump had to be boosted in order to simulate the kick that one would feel from a real crossbow. The user interface clicks were created using a ballpoint pen. Audacity allows you to export to multiple different file formats. I decided to export my sounds to waveformat with 44.1kHz compression, which is a regularly used in video games.

6 LMMS

6.1 LMMS user interface

LMMS user interface is similar to a lot of professional music making software. It has a lot of similarities with the commercial counterparts that are used within the music industry. (LMMS, 2020) On the left side you have multiple tabs with different sample sounds that you can use to make songs with. The middle area contains three separate windows when you start a new project. The first one is called the song-editor, in here you can drag your sound samples as layers. You can then choose if you want to add notes or create a beat or a bassline layer. To add notes, you double click on the note layer and it opens up a new window with a piano in it. By pressing the pen tool above the piano, you can add notes and by pressing the erase button, you can delete notes. The notes can also be stretched to different lengths. In similar fashion, if you double click on the beat layer, it opens up a new window where you can edit your beat by dragging samples in and pressing squares to toggle if a sample should be played or not. The second window contains your effects that you can use to mix your music with. Like in the sound-editor window, you can add multiple effect layers to get a desired sound. The library contains a lot of different effects for you to choose from. The effects on unix systems are shared object files, which are dynamically linked libraries. The third window is the controller rack where you can place controllers. The controllers enable you to automate the values of knob, slider and other controls. On the top you have a metronome, tempo, time signature and tools that open different work windows for you.

6.2 Composing music

The music track for the game is a short loop that is interesting enough to keep the player focused on the game but complex enough to not be annoying. The track consists of three different layers. The first layer is a bass sample that makes up the main melody. By adjusting the pitch a little bit higher, you get a sound that suits as a background melody. The second layer is a drum loop sample that scales with the tempo. The pitch and bass have been adjusted, so that it is not louder than the main melody. The third layer contains a custom drum beat to add a little bit more complexity to the music track. It is constructed by having an acoustic bass drum, opened hi-hat cymbal and a snare. To make the music sound intense, the tempo is set to 150 beats per minute with 4/4 time signature.

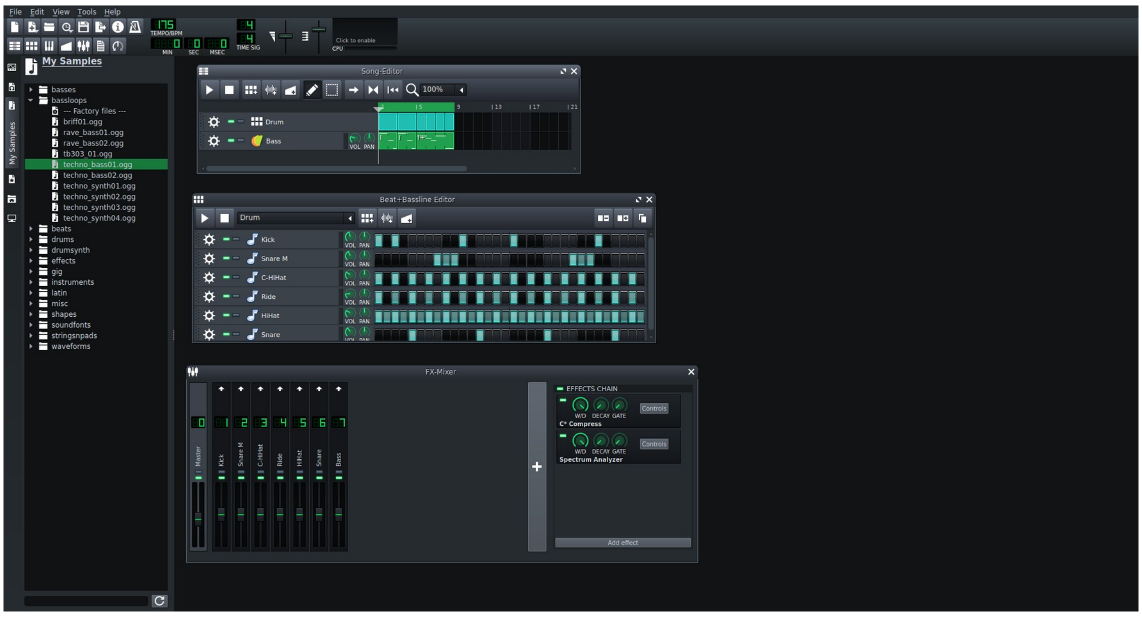


Figure 12. LMMS user interface

7 GODOT

7.1 Godot 3.2 user interface

Godot has a very friendly user interface. The main frame has been divided into multiple different sections. On the left you have the node hierarchy tree, where you can place all of you game items and save it as a scene. You also have the project file explorer, where you can create new Godot specific resources and organize your files and folders. On the right you have the node inspector panel. In there you can see all the options for a selected node from the node hierarchy tree. At the bottom you can find the console output, debugger, audio settings and Godot’s built in animation tool. At the top you have your default tools and settings for the editor and the project but also the tabs for both two-dimensional and three-dimensional scene editor, script editor and the public asset library. Unfortunately, this version of Godot does not support integrated development environment for C# in the script tab, so you will have to use an external integrated development environment like MonoDevelop or Visual Studio Code. However, if you are writing your game in GDScript, the script editor has a lot of powerful tools to help you on your programming journey. The game engine also allows you to change the layout completely all the way from panel positions to color themes. So, if you do not like how the scene node hierarchy tree is on the left, you can just drag and drop it to the right for instance.

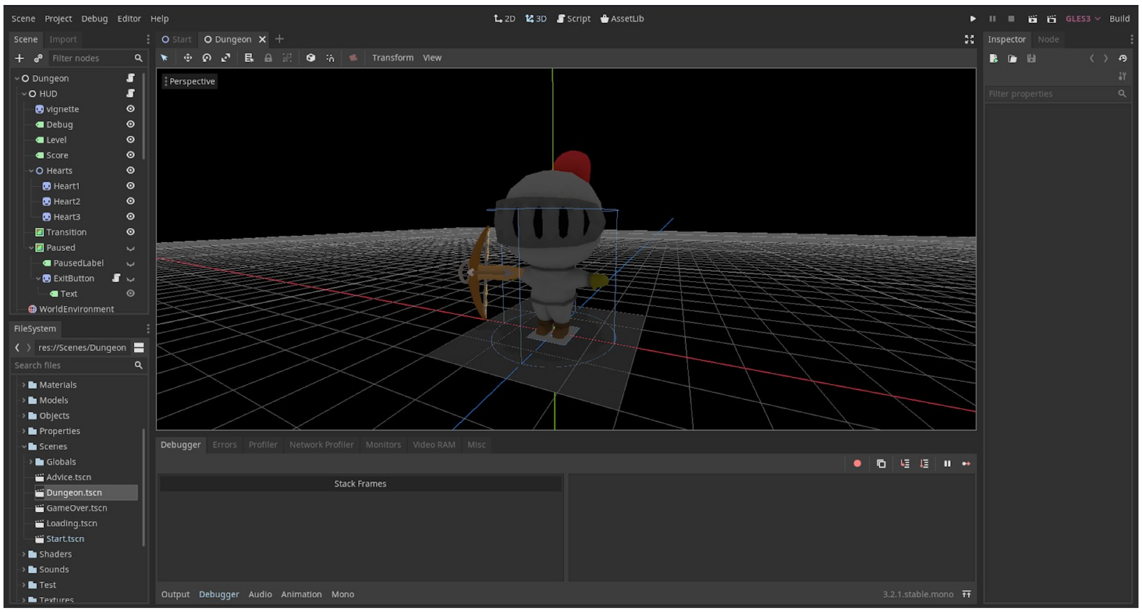


Figure 13: Godot 3.2 user interface.

7.2 Building the game

Godot game engine was built with a very specific object oriented design in mind. You divide the different sections of the games into something called scenes. These scenes contain objects that you can attach scripts to. The objects are usually referred to as nodes and the structure of the scene is called a hierarchy tree. With this kind of system it is relatively straightforward to setup a brand new project.

The first scene that loads when you start the game is the start menu. It contains two buttons: one that starts the main gameplay and one that shows you the instructions. The buttons do not change scenes as that would be unnecessary waste of resources, rather they just enable or disable other nodes in the hierarchy tree. The background and the buttons are two-dimensional sprite nodes that are configured to scale with the screen size and assigned image textures to the sprite nodes. In order to get some text on the buttons and on the other user interface inputs, a bitmap font had to be built inside Godot using C# code. The code essentially just maps the characters to a position on the bitmap texture as a two-dimensional vector and saves it as a Godot resource file. The user interface button class is also extended from the Godot default button class, so that it is possible to customize it with mouse hover and sound effects. The two-dimensional nodes in the hierarchy tree also have a depth index that is based on how high up or down low the node is or you can manually override it with your own number. The background has the lowest depth index and the logos and buttons on top of it have a higher depth index. When you press the play button, the player data default values are set and Godot handles the scene transition to the load scene that shows the temporary loading screen while a new dungeon is being generated.

The main dungeon scene contains a lot of different nodes with a lot of different purposes. There is an empty node that contains all of the two-dimensional head-up display child nodes: vignette, level label, score label, the heart sprites and the whole pause menu. The empty node has a script that handles all of the visuals of the head-up display. There is also a color rectangle node that has a custom shader that is used when the player enters and exits a dungeon. The shader uses a pre-rendered pseudorandom noise image generated in GIMP to save computing resources. Godot has made it easy to setup your scene environment. You simply create a world environment node and apply a new Godot environment resource to it. You can toggle between different options and change various values inside the environment object to get the desired look and feel. For example you can just with a press of a button enable screen space ambient occlusion. The player node is a kinematic body node, that contains a whole lot of child nodes. Few important nodes of the player are following: the collision shape that is needed with the kinematic body in order to detect and collide with other collidable bodies, a mesh instance with a material so we can see our player and an animation player that is needed to play the exported animations of the model. In order to get the crossbow to stay in the hand of the player, a bone attachment node is attached to the arm bone and a crossbow connected to it. This way when the player runs with the running animation, the crossbow follows the arm bone. To see what is going on in the game, we have to use a camera node that will follow the player with an offset vector value. The player mouse aiming works by ray casting a ray from the camera to the mouse point in three-dimensional space and returning the collision point of the ray. By rotating the player towards the collision point we get the desired gameplay effect. The arrows that are being fired from the crossbow when the player presses the left mouse button, are static entities with a script attached to them. The script makes the arrow rotate and translate forward with a constant speed. Godot has a built in feature, that enables you to put collision detections on different layers. This is important so that the arrows can collide with the imps and walls but no the stairs and the player. The collision layers are also used when ray casting from the camera to the three-dimensional scene to avoid unexpected rotations when aiming.

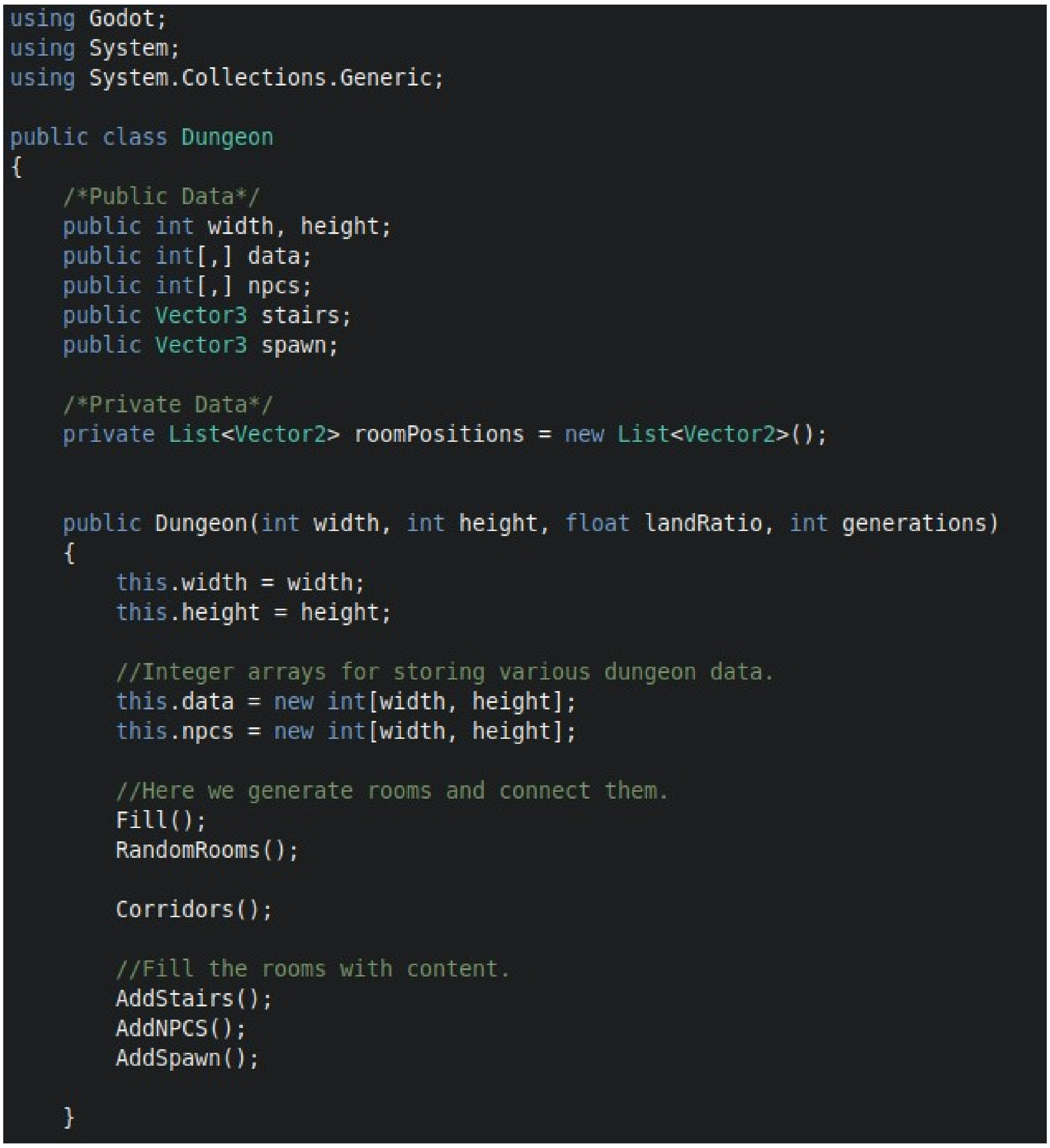
The root node of the scene contains a script that handles everything related to the dungeon. It starts by generating the necessary data and stores it in a dungeon class object. To construct the dungeon, a set of three-dimensional tile models are loaded. The script iterates through the dungeon cells and chooses a tile model from the set, which is based on the adjacent cells of the current cell. There are ten different models to choose from as they are already rotated in Blender to make the optimization logic part easier. The models have been modeled to fit together with each other, thus hiding any sharp seams. Even though Godot has a system to batch individual nodes and materials together to reduce draw calls, it is not considered a good practice to place each model individually as an instance to the scene. Instead, it is advised to create a single model or a few larger models by clamping the smaller models together. This is done by using Godot’s surface tool class. With the specified tool, you can take a model and all of its data, including vertices and uv maps, clamp them together and generate a new mesh, as well as normals. The code that builds the dungeon, splits the map into smaller chunks and uses the surface tool class to build the mesh. When it has finished building the mesh, it creates a new collision shape and generates a concave polygon shape for it, so that objects of the game have boundaries. The system then proceeds by adding the chunk in the scene and changes its translation accordingly, so that it lines up with the other chunks of the dungeon.

Before adding any non-playable characters to the scene, a navigation system is created with Godot’s built in classes and tools. It works by having a navigation node in the scene and a navigation mesh as a child node. One can then order a navigation agent to move by calling functions from the navigation node that return path data. If you have a static scene, you can use the user interface of Godot to generate a navigation mesh for the navigation node. However, the practical part of this thesis generates new content every time a player enters a dungeon. This is solved by generating a navigation mesh runtime after the system has generated the dungeon chunks. It loops through each floor cell and places a plane mesh with the size of a single tile from the set and makes it a single mesh with Godot’s surface tool class. Once the mesh has been built, it then proceeds by calling a function that creates a navigation mesh from the mesh built by surface tool.

When the code has reached to the point where it has generated all the dungeon chunks and the navigation mesh, it continues by adding the non-playable characters into the dungeon by iterating through the array in the dungeon class that contains information about what kind of non-playable characters exists in there. The non-playable characters are node trees that are stored on the disk as a Godot resource. They are loaded in runtime with the resource loader class and when the computer has found a non-playable character in the array, it creates a new instance of the non-playable character and positions it accordingly. The non-playable characters are constructed and programmed with an object oriented structure in mind. There is a main class, that contains the essentials for the creatures that inhabit the dungeon. This class contains for example the model, the dissolve shader, hearts, speed of the character, whether it is hurt or dead and other important variables that make the non-playable character work. There are two different extensions of that specific base class in order to create different distinguishing gameplay elements: the enemy that attacks with a range attack and the enemy that attacks with a melee attack. Even though the gameplay mechanics are different for these two non-playable characters, they still share a lot of similar features regarding the artificial intelligence, both are navigation mesh agents with states. These states determine what functions are being called and what states the non-playable character can reach. The states are enumerations that being updated based on the variables and constants during the main game loop. Both imps have an idle state and a random walk state. These states are called when the player is not near the imps. The idle state works with a timer. When the timer has hit a limit, the state is changed to random walk state. The random walk state selects a random point nearby and walks to it and when the imp has reached the goal it goes back to idle state again. By giving a minimum and maximum value to the timers and walk ranges, the non-playable characters feel more natural. The imps also have a constant for reaction distance, which is the distance when they starts moving towards the player and break away from their idle state. The current distance to the player is then measured and constantly compared to the reaction distance constant. When the current distance has been hit, the imps change their state to chase state. The imps differ with the attacks, they both have a constant for attack distance, the ranger imp has a longer attack distance so that it has space to fire its spell and the melee imp has a shorter attack distance so that it has to run all the way up close to the player. When the imps have reached the attack distance, they change their state from chase to attack. That is when a function is called that either fires a spell or directly hurts the player. The animations are also bound to the states, for example when the imp is just randomly walking, the walk animation is playing but when the imp is chasing the player, the run animation is playing.

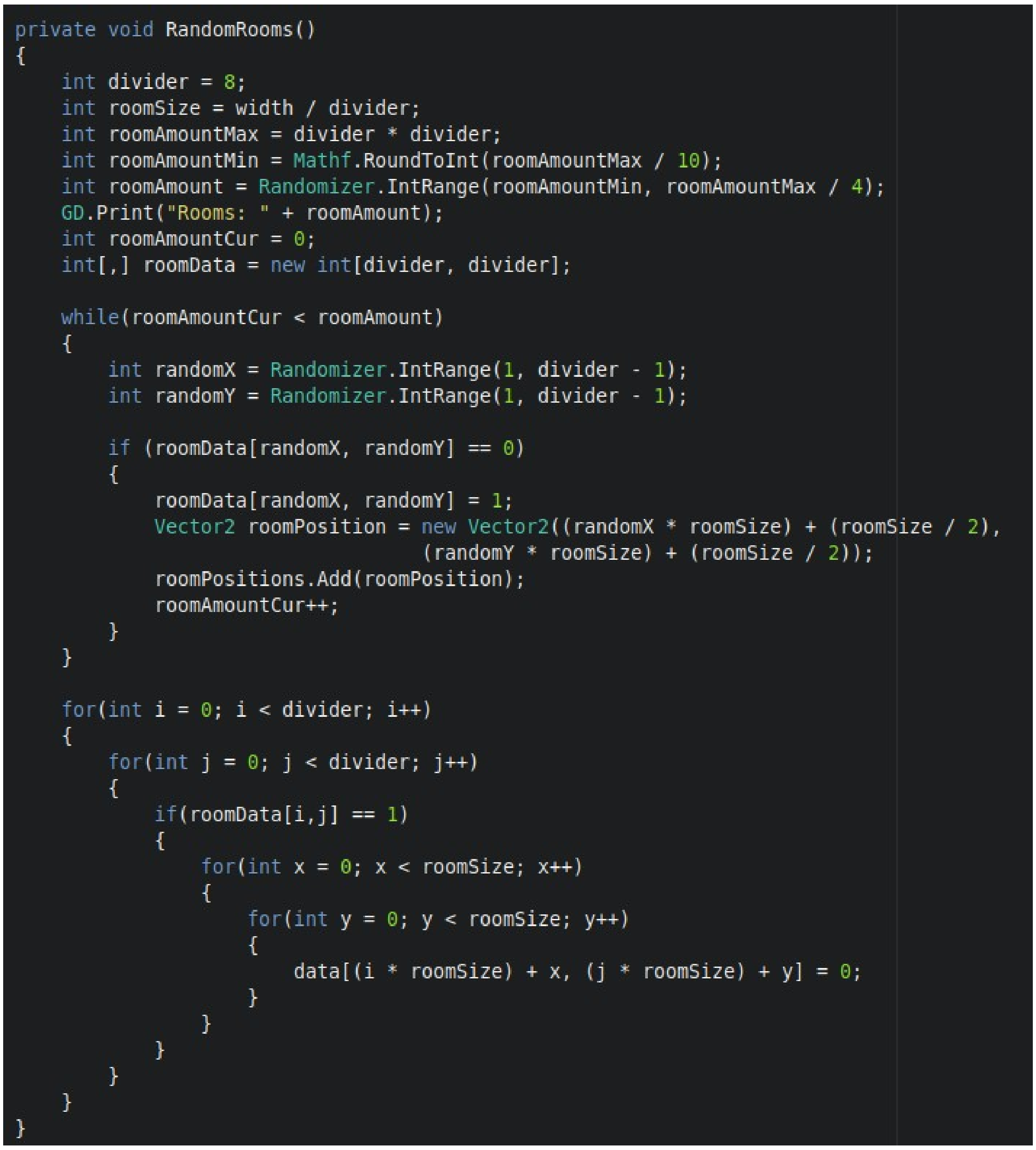
To progress to the next level the player has to touch the staircase, which is an area node with a collision shape. When the player enters the collision shape, the game is paused with Godot’s pause mode function so that enemies cannot hurt you while you are waiting and the transition progress starts. The transition is handled by referring to a variable in the shader that has been added to the color rectangle in the heads up display node. This is a variable that scales from zero to one and it is the value that determines how much the noise mask is cut off. When the cut off variable has reached maximum value, the code increments the level variable and loads the loading scene, which loads a new dungeon. The cut off variable will be now decremented to zero and when it has hit the minimum value, the game gets unpaused and the gameplay can continue.

7.3 Relevant C# code



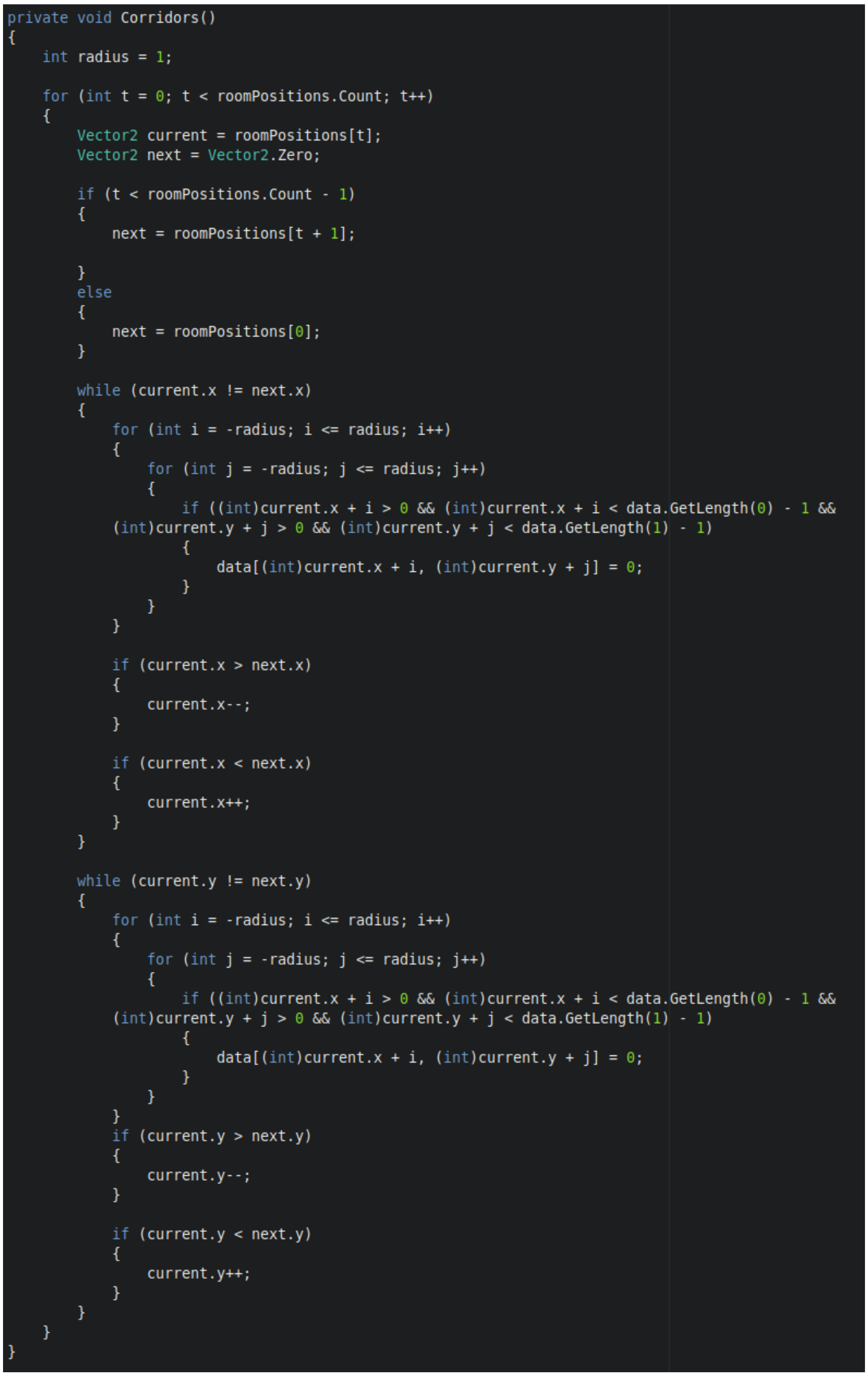
```csharp
using Godot;
using System;
using System.Collections.Generic;

public class Dungeon
{
    /*Public Data*/
    public int width, height;
    public int[,] data;
    public int[,] npcs;
    public Vector3 stairs;
    public Vector3 spawn;

    /*Private Data*/
    private List<Vector2> roomPositions = new List<Vector2>();


    public Dungeon(int width, int height, float landRatio, int generations)
    {
        this.width = width;
        this.height = height;

        //Integer arrays for storing various dungeon data.
        this.data = new int[width, height];
        this.npcs = new int[width, height];

        //Here we generate rooms and connect them.
        Fill();
        RandomRooms();

        Corridors();

        //Fill the rooms with content.
        AddStairs();
        AddNPCS();
        AddSpawn();

    }
```

Figure 14: Dungeon class constructor



```
private void RandomRooms()
{
    int divider = 8;
    int roomSize = width / divider;
    int roomAmountMax = divider * divider;
    int roomAmountMin = Mathf.RoundToInt(roomAmountMax / 10);
    int roomAmount = Randomizer.IntRange(roomAmountMin, roomAmountMax / 4);
    GD.Print("Rooms: " + roomAmount);
    int roomAmountCur = 0;
    int[,] roomData = new int[divider, divider];

    while(roomAmountCur < roomAmount)
    {
        int randomX = Randomizer.IntRange(1, divider - 1);
        int randomY = Randomizer.IntRange(1, divider - 1);

        if (roomData[randomX, randomY] == 0)
        {
            roomData[randomX, randomY] = 1;
            Vector2 roomPosition = new Vector2((randomX * roomSize) + (roomSize / 2),
                                       (randomY * roomSize) + (roomSize / 2));
            roomPositions.Add(roomPosition);
            roomAmountCur++;
        }
    }

    for(int i = 0; i < divider; i++)
    {
        for(int j = 0; j < divider; j++)
        {
            if(roomData[i,j] == 1)
            {
                for(int x = 0; x < roomSize; x++)
                {
                    for(int y = 0; y < roomSize; y++)
                    {
                        data[(i * roomSize) + x, (j * roomSize) + y] = 0;
                    }
                }
            }
        }
    }
}
```

Figure 15. Room generation function



```
private void Corridors()
{
    int radius = 1;

    for (int t = 0; t < roomPositions.Count; t++)
    {
        Vector2 current = roomPositions[t];
        Vector2 next = Vector2.Zero;

        if (t < roomPositions.Count - 1)
        {
            next = roomPositions[t + 1];

        }
        else
        {
            next = roomPositions[0];
        }

        while (current.x != next.x)
        {
            for (int i = -radius; i <= radius; i++)
            {
                for (int j = -radius; j <= radius; j++)
                {
                    if ((int)current.x + i > 0 && (int)current.x + i < data.GetLength(0) - 1 &&
            (int)current.y + j > 0 && (int)current.y + j < data.GetLength(1) - 1)
                    {
                        data[(int)current.x + i, (int)current.y + j] = 0;
                    }
                }
            }

            if (current.x > next.x)
            {
                current.x--;
            }

            if (current.x < next.x)
            {
                current.x++;
            }
        }

        while (current.y != next.y)
        {
            for (int i = -radius; i <= radius; i++)
            {
                for (int j = -radius; j <= radius; j++)
                {
                    if ((int)current.x + i > 0 && (int)current.x + i < data.GetLength(0) - 1 &&
            (int)current.y + j > 0 && (int)current.y + j < data.GetLength(1) - 1)
                    {
                        data[(int)current.x + i, (int)current.y + j] = 0;
                    }
                }
            }
            if (current.y > next.y)
            {
                current.y--;
            }

            if (current.y < next.y)
            {
                current.y++;
            }
        }
    }
}
```

Figure 16: Corridor generation between rooms

8 Conclusion

When you are creating a game, you have to consider a lot of different aspects. It is not only the economy and budget that you should focus on but also the features of the different software and the overall workflow. Having a product that is too difficult to use or too expensive, will disrupt the workflow and in the worst case stop it completely. The cost is usually not an issue with free open source software but the overall workflow might suffer because of some design decisions that have been made for a specific program. However, the programs that are presented in this thesis are all very intuitive and responsive to work with. As one can note in this thesis, they all include a large variety of features that a lot commercial counterparts offer, thus making them great alternatives for game development and other purposes. This can clearly be seen in the practical part of this thesis. It checks all the marks that were set: a three-dimensional world with textured and animated models, background music and sound effects to help with the visuals and gameplay that gives psychological rewards. These are all elements that exists in modern games.

Blender covers most of your basic and advanced model making needs. You can create meshes, texture them and give them animations. The community around the software is huge, so if you get stuck and need help, the internet is full of tutorials, guides and help threads. Blender is also highly suitable for game development since you can export models to multiple different formats by default, which makes it very flexible to create content for different game engines that support different model formats.

With GIMP one can easily create textures and two-dimensional art. The program is loaded with tools that you can manipulate pixel data with. All the way from beginner paint tools to intermediate noise and fractal functions, GIMP covers a big area of features that makes it very suitable for game development. The program supports drawing tablets as well, making it easier for artists to create content. Just like Blender, GIMP can import and export a lot of different file formats, which makes it a very valid candidate for creating visuals.

Creating sound effects in Audacity is straightforward as it gets. The minimalistic user interface might fool a user about the complexity of the software. The more advanced tools and settings are available under submenus. At the current state, Audacity is not meant for making music, it supports MIDI files but it is not very user friendly.

The user interface of LMMS might first seem a bit confusing but once you have learned how it works, the program shows its true potential. Composing music becomes efficient and easy. While the software has a big audio sample library, a lot of them are very similar and some of them are low quality. However, you could get better sounding samples somewhere else and still use the powerful tools that LMMS provides to compose music. LMMS is more than suitable for music production and thus suitable for game development.

Godot game engine in its current state is more than capable of producing interactive content. It is not a very old game engine but still capable of rendering both two-dimensional and three-dimensional games flawlessly. The large diversity of available nodes makes it possible to with imagination to create different themed video games with a lot of different mechanics. The engine is very modern, as it supports modern shading languages and render backends and has multiple different platforms that you can export to.

As a final word, jumping between the different chosen open source software did not slow down the creation progress and they all had a lot of features, making it possible to create content of different kind. This thesis has proven that studios, whether they are big or small, independent developers and anybody that is interested in developing a fully-fledged video game, can choose open source software.

APPENDIX - SVENSK SAMMANFATTNING

1.1 INTRODUKTION

Examensarbetets huvudsakliga mål är att forska om programvara baserad på öppen källkod tillämpar sig för utveckling av datorspel baserat på arbetsflöde och funktionalitet. Arbetet jämför inte gratis programvara med kommersiell, utan fokuserar sig på skapandet som process och funktionaliteter. Arbetet går inte heller djupare in på grafisk design, tredimensionella modellerings principer eller teori av musik och ljudeffekter. Den praktiska delen av examensarbetet är ett spel med tredimensionell grafik, algoritmer, ljudeffekter och musik.

1.2 PROGRAMVARA

Programvaran som har valts för den praktiska delen av examensarbetet, är de program som är baserade på öppen källkod och som jag personligen anser att används i stor utsträckning. Personlig preferens har även påverkat val av programvaran.

1.3 ARBETSFLÖDE

Med Blender skapas de tredimensionella modeller samt ges animationer för den praktiska delen av examensarbetet. Modellerna skapas med att börja från en enkel primitiv modell som t.ex en kub eller sfär och sedan lägger till mera polygoner efterhand. Med att veckla ut modellens polygoner på ett tvådimensionellt plan, kan man ge texturer som sedan kan ses på modellens yta. För att animera modellen, skapar man ett skelett som man binder ihop med polygonernas hörnpunkter med olika tyngder. Skelettets olika ben kan ges punkter på en tidslinje, som bestämmer benets translation och transformation. Då man itererar genom punkterna, spelas animationen upp med att linjärt interpolera igenom translationerna och transformationerna.

Med GIMP kan man skapa texturer och annan grafik. För att spelets visuella område är byggt av tredimensionella klossar som passar ihop med varandra, bör de ha texturer som är sömlösa, dvs. texturer man kan placera bredvid varandra och det syns inte några kanter eller konturer. Detta kan man göra med att rita med symmetri pensel verktyget i programmet. Användargränssnittet skapas också i GIMP, de är tvådimensionella bilder som läggs ovanför spelvärlden under spelets omlopp.

Audacity används för att banda och redigera ljud med. Ljudeffekterna för spelet skapas med att banda in ljud med en kondensatormikrofon. Därefter redigeras ljuden med att ändra på olika inställningar och med att lägga till effekter. Man kan t.ex ändra på ljudets tempo, tonfall, bas och efterklang.

För att komponera musik, används LMMS med ljudbiblioteket som kommer med programvaran. Man kan skapa olika lager av instrument och ha olika lager för takter och melodi. Musiken för spelet skapas med att ha trummor på ett lager och en bas med förhöjd ton som melodi lager.

I Godot spelmotorn knyts allting ihop med C# programmering. Systemet fungerar med någonting som kallas för noder. Noder är klasser som man kan föra över till en scen och för att kunna växelverka med noderna kan man lägga till C# kod till dem. Spelet fungerar med att ha en scen som startar spelets omlopp och lägger de värden som behövs till förinställda värden. Fängelsehålan som spelet spelar sig i, byggs upp med en algoritm som placerar ett antal rum på måfå och kopplar ihop dem med korridorer. Dessa rum och korridorer byggs upp av färdiga tredimensionella modeller som har exporterats från Blender med texturer. För att kunna lägga fiender på spelplanen, måste man generera en navigerings modell som fiendens programkod använder sig av för att få gångbanor för deras mål. För att spelet kan köras utan några större problem bör man även optimera byggnaden av fängelsehålor och hantering av fiender. Spelområdet är delat i mindre delar som laddas enligt distans till spelaren.

1.4 Resultat

Resultatet som examensarbetet nådde var att det bevisade att det är möjligt att skapa datorspel med programvara baserad på öppen källkod. Det förekom inte några problem med att hoppa mellan programmen och skapa någonting nytt. Arbetsflödet påverkades inte heller på negativt sätt eftersom alla program som användes i examensarbetet hade flera olika verktyg och redskap med intuitiv användargränssnitt, vilket möjliggjorde att man kunde snabbt och effektivt skapa någonting nytt. Det här leder till det första frågan som introducerades: är det möjligt för mindre spelbolag och oberoende spelutvecklare att använda sig av programvara som är baserad på öppen källkod? Examensarbetets praktiska del bekräftar att det är fullständigt möjligt att bygga ett datorspel med dessa verktyg och teknik som uppfyller kraven som har definierats för den praktiska delen. Den tredimensionella grafiken med modeller och animationer tillsammans med partiklar och skuggor skapar en visuell upplevelse som de flesta moderna datorspelen strävar efter. Ljudeffekterna och musiken stöder den visuella upplevelsen och spelets omlopp. Sammanlagt skapar de en helhet som kan anses modernt och bevisar att mjukvaran är kapabel för seriös produktion.

References

3D performance and limitations - Godot Engine. Available at:

https://docs.godotengine.org/en/stable/tutorials/3d/
3d_performance_and_limitations.html

Accessed 04.05.2020.

Collection: Game Engines, GitHub.

Available at: https://github.com/collections/game-engines

Accessed: 11.05,2020.

Toftedahl, M., 2019, *Which are the most commonly used Game Engines?*. Available at:

https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/
Which_are_the_most_commonly_used_Game_Engines.php

Accessed 11.05.2020.

Wang, H. & Sun, C-T., 2011, *Game reward systems: Gaming experiences and social meanings*. Available at:

http://www.digra.org/wp-content/uploads/digital-library/11310.20247.pdf

Accessed 11.05.2020.

LMMS, 2020. Available at:

https://lmms.io/

Accessed 11.05.2020.

Xinogalos, S., 2017, *Overview and Comparative analysis of Game Engines for Desktop and Mobile Devices.* Available at:

https://www.researchgate.net/publication/322027338_Overview_and_Comparative_Analysis_of_Game_Engines_for_Desktop_and_Mobile_Devices

Accessed 11.05.2020.