jamk.fi

Development of a 3D mahjong video game in Godot Engine

Henri Mäkelä

Bachelor's thesis
May 2021
Information and Communications
Degree Programme in Information and Communications Technology

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Mäkelä, Henri	Bachelor's thesis	May 2021
	Number of pages 69	Language of publication: English
		Permission for web publication: Yes

Title of publication

Development of a 3D mahjong video game in Godot Engine

Degree programme

Information and Communications Technology

Supervisor(s)

Immonen, Jani; Salmikangas, Esa

Assigned by

Abstract

More and more games these days are developed using third-party game engines, as they help to save company resources. When it comes to choosing the right engine for the right project, the decision may be a difficult one, depending on many factors such as project scope and desired visual fidelity for the graphics. One of the currently emerging game engines is Godot Engine, the engine of choice for an increasing number of developers.

However, its status and capabilities especially in 3D development are still relatively unexplored compared to other prevalent engines like Unity. Therefore, the goal was to determine the viability of using Godot Engine for developing 3D games. In order to accomplish this, an in-depth case study was carried out by using the engine to develop a three-dimensional Japanese riichi mahjong game. Mahjong is an existing turn-based table game akin to poker, albeit more complex and suitable for a 3D case study.

The programming language used during development of the project was .NET Framework's C#. Among Godot's tested functionalities were processes such as game world creation, user interface implementation, shader writing, and game distribution. A multiplayer mode with a working server and client architecture was also implemented, making use of the engine's networking features. The development process was fully documented, and problems were presented and discussed on the way.

Godot proved to be an able game engine for the purpose of developing 3D games the size of mahjong. A functional prototype of the game was developed without major problems. Highlights from the results showed strengths in the engine's scene system and the ease of implementing multiplayer. However, its node-based workflow takes time getting used to when developing with C#, and some of the results such as engine performance might not be applicable to larger game projects. Based on the results, Godot is still a good alternative to other 3D engines such as Unreal Engine or Unity, when it comes to 3D games of smaller caliber.

Keywords/tags

video game, multiplayer, mahjong, riichi, godot, 3D, game development, game engine

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Mäkelä, Henri	Opinnäytetyö, AMK	Toukokuu 2021
	Sivumäärä 69	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: Kyllä

Työn nimi
3D mahjong pelin kehitys Godot Engine -pelimoottorilla

Koulutusohjelma
Tieto- ja viestintätekniikka

Työn ohjaaja(t)
Immonen, Jani; Salmikangas, Esa

Toimeksiantaja(t)

Tiivistelmä

Yhä useampi peli näinä päivinä kehitetään kolmannen osapuolen pelimoottorien avulla, koska niiden käyttö edesauttaa yritysten resurssienhallintaa. Oikean pelimoottorin valinta projektia kohden voi kuitenkin osoittautua hankalaksi riippuen monista eri tekijöistä, kuten projektin laajuudesta ja grafiikoiden tavoitellusta laadusta. Yksi parhaillaan nousevista pelimoottoreista on yhä useamman pelikehittäjän suosima Godot Engine.

Moottoriin kohdistuvat tutkimukset sen asemasta ja kyvykkyydestä tuottaa 3D pelejä ovat kuitenkin jääneet vähemmälle, etenkin verrattuna muihin esillä oleviin moottoreihin kuten Unityyn. Näin ollen tutkimuksen tavoitteeksi valikoitui selvitys Godot Enginen 3D-pelituotannon kyvykkyydestä. Kyvykkyys määriteltiin toteuttamalla perusteellinen tapaustutkimus, jossa kehitettiin kolmiulotteinen Japanilainen riichi mahjong -peli moottoria käyttäen. Mahjong on pokeria muistuttava vuoropohjainen pöytäpeli mutta sitä monimutkaisempi sekä sopivampi 3D tapaustutkimukseen.

Pelikehityksen aikana käytetty ohjelmointikieli oli .NET Frameworkin C#. Godotin testattuihin toiminnallisuuksiin kuuluivat prosessit kuten pelimaailman luonti, käyttöliittymän implementointi, varjostimien kirjoitus ja pelin julkaisu. Pelille kehitettiin myös toimiva asiakas-palvelin -arkkitehtuurin mukainen moninpelitoiminnallisuus käyttäen hyödyksi moottorin verkkopeli ominaisuuksia. Kehitysprosessista tehtiin täysi dokumentaatio, jossa myös esitettiin ja käsiteltiin kehityksen aikana kohdattuja ongelmia.

Godot osoittautui varsin eteväksi pelimoottoriksi mahjongin kokoisten 3D-pelien kehitykseen. Toimiva prototyyppi pelistä saatiin kehitettyä ilman suurempia ongelmia. Tulokset korostivat vahvuuksia moottorin skene-systeemissä sekä moninpelin toteutuksen helppoudessa. Godotin solmu-pohjainen työnkulku vaatii kuitenkin totuttelua, kun kehitys tehdään C#-kielellä. Myöskään joitain tuloksia, kuten moottorin suorituskykyä, ei välttämättä voida soveltaa laajempiin peliprojekteihin. Godot on kuitenkin pienemmän mittakaavan peliprojekteissa hyvä vaihtoehto muille 3D moottoreille kuten Unreal Enginelle ja Unitylle.

Avainsanat
peli, moninpeli, mahjong, riichi, godot, 3D, pelinkehitys, pelimoottori

Muut tiedot

Contents

Figures

1 Introduction

In the video game industry, many people dream of creating their own game engine. Large blockbuster game studios are especially known for using in-house engines in their AAA games. Admittedly, there are many advantages to developing games on tailor-made engines. When given the opportunity to implement engine features whenever needed, it's possible for developers to take complete control and achieve any level of performance or graphical effect in a game that they require. However, this doesn't come without caveats. Engine development is incredibly time-consuming and requires a lot of motivation, resources, and technical know-how in order to succeed. Therefore, many developers nowadays, especially smaller and less experienced teams, opt to use third-party game engines instead. One of those engines is Godot Engine—an open-source alternative that's been gaining a considerable amount of attention during recent years. While its community has been steadily growing, conversely not much research has been done on its game development capabilities, when compared to more prevalent engines like Unity or Unreal Engine.

As a rising competitor in the field of game engines, Godot and especially its performance in 3D has started to pique the interest of developers. 3D is a hot topic in the game industry, as it can prove to be challenging for many game engines to implement, while still posing technical problems for GPU manufacturers to solve. Although Godot has received praise for its capacity in 2D game development, its ability to tackle 3D rendering is less touched upon, and could benefit from more academic research. Therefore, Godot's viability in the development of 3D games was chosen as the subject of this study. The best method for evaluating such viability, is generally by means of developing a game project with the engine in question. This is a form of case study, where the engine's features and performance are put to the test, and assessed separately to form a conclusion. For the results to be deemed reliable, care should be taken in picking the right type of game for the project, while taking time, scope and available resources into account.

Thus the method chosen for the case study was the development of a Japanese mahjong game in Godot Engine. Mahjong features tile-based multiplayer gameplay, making it a fitting option for a case study, as its characteristics can easily be taken advantage of in a three-dimensional video game world. It's also modest enough in size to be developed by a single person without prior experience with the game engine, as is the case here. I'm also passionate about the game, with the intention of later continuing development and eventually expanding the game into a real product, if it were to turn out that Godot is a good platform for doing so. In summary, the goal of the study was to create a working prototype of a 3D mahjong game, and to assess Godot's features and performance in the process, forming a conclusion of its capability by the end of it.

2 Game engines

Game engines are frameworks for creating video game software. They can be separated into components (see Figure 1) such as the rendering engine, physics engine, audio engine, and the main program containing the game loop (Baker, 2016). Engines come packed with a wide variety of features necessary for game development, which help to save both time, capital and manpower for companies. These include features like animation, artificial intelligence, collision detection, memory management, and networking. Although game engines have many tools to choose from, they don't have a defined standard architecture, so not all engines necessarily come with the same features. (Zarrad, 2018.) Distribution of games to other platforms is another strong point of using game engines, as they allow building games for multiple platforms with ease, including consoles, mobile devices, PC operating systems, and the web. This allows developers to reach larger audiences with their games, potentially increasing their revenue. Otherwise, another option is to send the game to a porting company that's specialized in porting games to other platforms, which of course comes with a price.

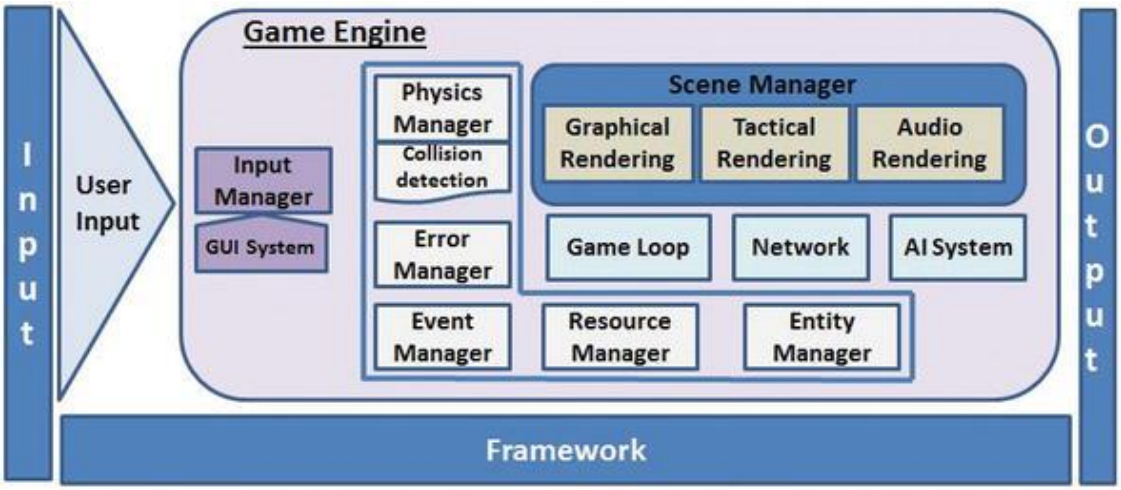


Figure 1. Components in game engine architecture. From “Game Engine Solutions” by Zarrad, A., 2018, p. 79, Figure 2 (http://dx.doi.org/10.5772/intechopen.71429). CC BY 3.0.

Based on availability, game engines can be generally divided into two main groups: in-house engines and third-party engines. In-house engines are proprietary pieces of software developed by game companies themselves, such as Valve Software’s Source 2 engine. In-house engines can represent decades of engineering work by a single company, and over the years, may form an entire evolution tree of different versions based on each other. On the other hand, third-party engines can either be company or community-made, and are generally available to the public for free, or through a license fee. Sometimes companies create new third-party engines by releasing publically available branches of their in-house engines, while keeping a more state-of-the-art branch of the engine for themselves. Business models for engines can therefore change over time, like with Crytek’s CryEngine, which started as an in-house engine, then became free, and finally adopted a 5% revenue share plan (Batchelor, 2018).

At other times, the engine might be released to the public completely for free and become a community project, like in the case of Godot Engine. In addition to Godot, some other examples of popular third-party engines include Unity, GameMaker, Phaser, and Unreal Engine. All of them come with a vast set of tools for developing games, and they all have documentation and resources provided by their communi-

ties. (Schardon, 2021.) Having been developed since 2005, Unity (see Figure 2) is arguably the most popular game engine right now, which can be seen by the vast amount of content featured in its asset store (Schardon 2021; Stephenson 2019). In 2019, under one-third of game studios in the UK were using proprietary in-house engines for their projects, while the rest were using third-party ones (Stephenson, 2019). By choosing to use a third-party engine rather than developing their own, there's no need for developers to invest time in learning complex mathematical algorithms and advanced design patterns, which are the building blocks of these engines.

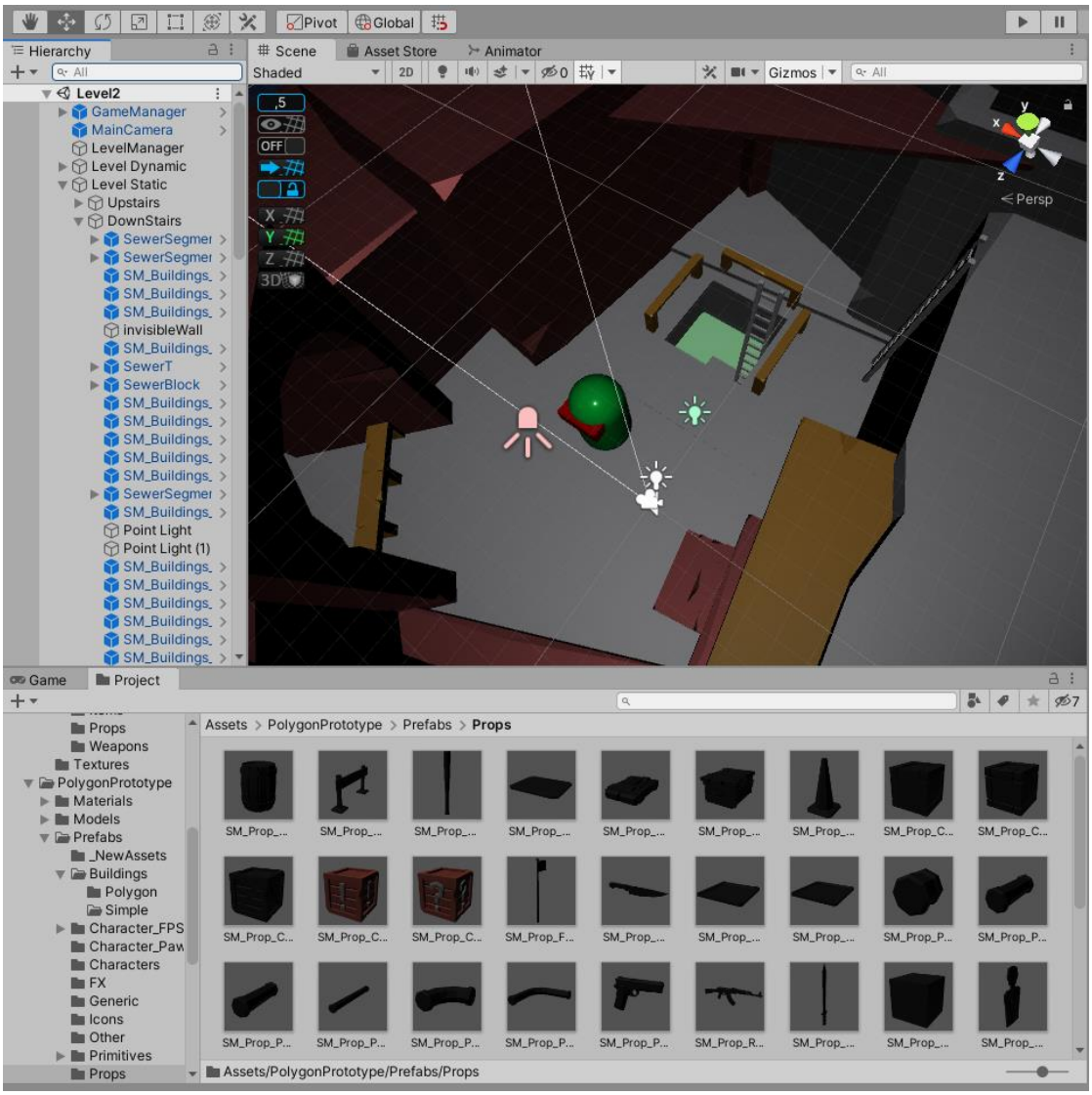


Figure 2. Game development in the Unity game engine. Unity. Retrieved on 22 May 2021.

It naturally follows that one of the downsides of choosing a third-party engine can be the uncertainty of whether it's able to fulfill the technical requirements that the developers have set out for their project. Engines come in all varieties of licenses and intended use cases, and they all have their own limitations, so with those points in mind, it's important to choose the right engine for the right project. Every project should be evaluated separately in order to pick the best engine for the job. Not only desired features and visual effects, but also attributes such as ease of learning, distribution platforms, developer experience, and licensing policies should be taken into account. As Lavieri (2018, chapter 1, para. 5) effectively summed it up: "Selecting the right game engine for your game project is a critical pre-development step. Not every game engine will work for your game, and no single game engine will work for every game."

For example, for those who are still learning game development and seek to create a simple 2D game that can be played on the browser, Phaser is a viable choice. If the development team is working on a more serious project but doesn't come with the most knowledgeable people on programming, GameMaker does it better. If the aim of the game is to be more realistic, most likely three-dimensional and coupled with flashy effects, then a contemporary 3D game engine like Unity or Unreal Engine is probably most ideal, provided that the development team has experience in the field. It can be assumed that the capabilities of an engine can be defined based on the types of games they're known for. However, while it's beneficial for the industry to develop the capabilities of game engines, it's also important to research and document them so that such assumptions don't have to be made. This is especially applicable to engines that don't yet have a long list of games under their belt, such as Godot Engine.

3 Mahjong

3.1.1 Status

Mahjong is a 4-player tile-based game originating from China. With its unpredictable outcomes and probability-based strategy, it's considered a game of both luck and skill, therefore satisfying the definition of an imperfect information game. Its complexity also makes it a good subject for self-learning AI research. (Spencer 2019; Gao et. al. 2019, 1-2.) Since its invention in the late nineteenth century, its popularity has increased and spread to other Asian countries such as Japan, and also to other continents of the world including Europe and North America. Nowadays it's played both socially and professionally across Eastern Asia (Spencer 2019). In the course of time, these regions have developed their own elaborate rulesets and standards for the game, such as American mahjong, classic Hong Kong mahjong, and Japanese riichi mahjong (see Figure 3), the latter which is especially popular in video games.

What makes riichi mahjong more popular to its variants in this regard, is its strategic depth and greater reliance on skill as opposed to luck, which adds a competitive edge to the game. Consequently, the game in this study will follow the better suited riichi ruleset as its foundation, together with some of the specific rules devised by the European Mahjong Association. Hereafter, the term "mahjong" will also be used to refer specifically to riichi mahjong. Finally, an important distinction should be made between riichi mahjong and the popular solitaire mahjong, which is a pair collecting singleplayer game that has nothing to do with the traditional type of multiplayer mahjong that the study focuses on.

Figure 3. A recently won hand in a game of real-life riichi mahjong (own photo). November 12, 2017.

3.1.2 Rules

A game of riichi mahjong begins by four players taking their seats, each seat in order representing a wind direction of north, east, south and west. If the seats aren't predetermined, the players will draw lots to determine the seating order as well as who gets to be east, also known as the dealer. Every player, using the shuffled tiles scattered along the table, then builds a wall that's two stacks high and 17 tiles wide. In automatic mahjong tables, this step is done automatically by the table itself, which is also the case for the game project. Two dice are then rolled by the dealer, which determine whose wall will be separated to distribute the starting tiles. The dice roller will count players counter-clockwise the amount shown by the dice; starting from themselves, then the seat right of them, then the one opposite, etc. The player thus determined will then separate their wall by counting tile stacks from right to left the same amount as indicated by the dice, making a small gap in the wall at the last stack. (European Mahjong Association 2016, 7.)

The seven tile stacks before the separation are known as the dead wall, which are usually considered excluded from the game, aside from the kan replacement tiles.

The dead wall's third top tile from the left is then flipped to reveal a dora indicator, "dora" being a bonus tile that awards more points to the players. The dora tiles are the ones that are next in consecutive order after the indicator—e.g., if the indicator is the seven of bamboo, the dora is then the eight of bamboo. In order to obtain their starting hands, players then draw 4-tile groups of tiles after each other, starting from the dealer, proceeding in clockwise order. After three groups, every player draws one more tile except for the dealer, who draws two, the latter draw simultaneously being their first draw of the game. The starting hands are then sorted, and the dealer will start the game by discarding a tile. (ibid., 8.)

Once the game has begun, players take turns to draw tiles from the wall in order to complete their hand before others. A turn begins by drawing a tile from the wall, and then ends by discarding the drawn tile or another tile from the player's hand. The turn order then moves counter-clockwise. Building hands is done by collecting groups of tiles that are generally made of 3-tile sequences or triplets, as well as pairs. A finished hand has 14 tiles and typically consists of four groups and a single pair. There are 136 tiles in total, with four of each tile. They consist of three suits: dots, bamboo and characters, which range from numbers 1 to 9, as well as honor tiles that are divided between winds and dragons. (ibid., 6.) It is also possible (depending on the situation) for players to complete groups by calling "chii", "pon" or "kan" on other players' newly discarded tiles, in which case the turn order may move and other players' turns may be skipped. Chii is used to complement a sequence (e.g., calling a 3 to 1-2 to get 1-2-3) and can only be called from the player that precedes the calling player (i.e. left side player).

Pon and kan, on the other hand, can be called from anyone, and are used to complement triplets and quadruplets, respectively. Making a kan will reveal an additional dora indicator, and allows the player to draw a replacement tile from the end of the dead wall, after which the last tile from the live wall is then appended to it. After calling, the player reveals the incomplete group from their hand, retrieves the called tile from the opponent's discard pile and sets the now completed group openly beside them. One of the group's tiles is then turned sideways to refer to which player the tile was called from (see Figure 4), after which the calling player

ends their turn by making a discard. Kan may also be declared by the player if they were to draw four of the same tile, which is referred to as a closed kan, and must also be displayed to other players. Calling tiles is an effective strategy to complete hands faster, but comes at the cost of lowering the value of most hands, and leaving less options for defensive play. (ibid., 9-12.)

Figure 4. Two tile calls made in a mahjong video game. A pon of green dragons from the right side player, and a chii of characters from the left side player. Retrieved on 21 May 2021.

After completing the hand, the winner is awarded with points according to its total value, calculated based on a series of winning patterns called “yaku”. They are akin to hands found in poker, although in mahjong, multiple yaku can be combined to increase the hand’s value even further. There are dozens of yaku featured in riichi, of which at least one is required to be eligible to win a hand. One of the most important ones, which the game also owes its name to, is riichi, which is the act of declaring that a hand is one tile away from being complete—a status also known as “tenpai”. Riichi is declared by saying “riichi”, discarding a tile sideways, and then placing a 1000-point wager stick in the middle, which can be regained by winning the round. If the round ends in a draw, the next player to win will gain the wager stick. Riichi, like many other yaku, can only be obtained with a closed hand—i.e. the player hasn’t called any tiles. After declaring riichi, the player may no longer change their hand composition aside from declaring a closed kan. Additionally, winning a riichi hand grants the winner access to uradora, which are extra dora indicators under the currently visible dora indicators, thus multiplying the amount of dora tiles. (ibid., 20.)

There are two ways to win: either calling another player's discarded tile as the winning tile, or by the player drawing the winning tile themselves, saying "ron" or "tsumo" respectively in each situation. Winning on an opponent's discard makes them liable for the whole sum of the hand's value, while self-drawing the winning tile splits the payment between players. However, winning on a discard is disallowed if the player has already discarded any of their current possible winning tiles, which is a rule known as "furiten"—one of the cornerstones of the game's defense strategy. All players typically start with 25000 points, and respectively lose or win more points by either paying for other players' won hands or winning themselves. The dealer wins and loses half as many points. Going negative on points, also known as "busting", ends the game prematurely. There are usually at least eight rounds, divided between east and south rounds. If no-one wins, the round is declared a draw, and the players that are in tenpai get a small amount of points from those that aren't. The seats (and dealer) rotate after every round, unless the dealer wins or the game ends in a draw with the dealer in tenpai, in which case the round is repeated. If the round is repeated or no-one wins, a counter stick ("honba") is added, with every stick contributing 300 points more to the next winning hand. Winning a hand clears the current amount of counter sticks. At the end of the game, the player with most points wins the game. Ties can be settled based on seat order or sharing the placement. (ibid., 16-17.)

3.1.3 Mahjong in video games

Following the game's vastly growing popularity in Asia, its spread into the video game industry was inevitable. Starting in 80's Japan, mahjong could already be found on the Nintendo Entertainment System, as a 2-player variation against a computer-controlled opponent for singleplayer convenience. Later in the same decade up until the mid-90's, the game also saw multiple releases for Japanese arcade machines. It took until the mid-2000's for mahjong to pick up in the form of browser games. Mahjong video games hadn't been able to reach the western market up until this point. One game in particular that received sizeable attention was Tenhou, a free-to-play online mahjong service developed by the Japanese software company C-EGG. Tenhou holds a large base of around 350,000 players, and is considered one of the

most popular mahjong platforms even today. It's also favoured by the western mahjong community as their platform of choice, partly due to browser plugin support that helps in localizing in-game menus. The free web-based version of Tenhou offers a simplistic pseudo-3D interface (see Figure 5) and also supports mobile devices. The subscription-based Windows client version, on the other hand, features full 3D graphics and more customization options. (Tenhou.net.) Tenhou serves as a good example of the desired end result for the Godot project's graphical look.

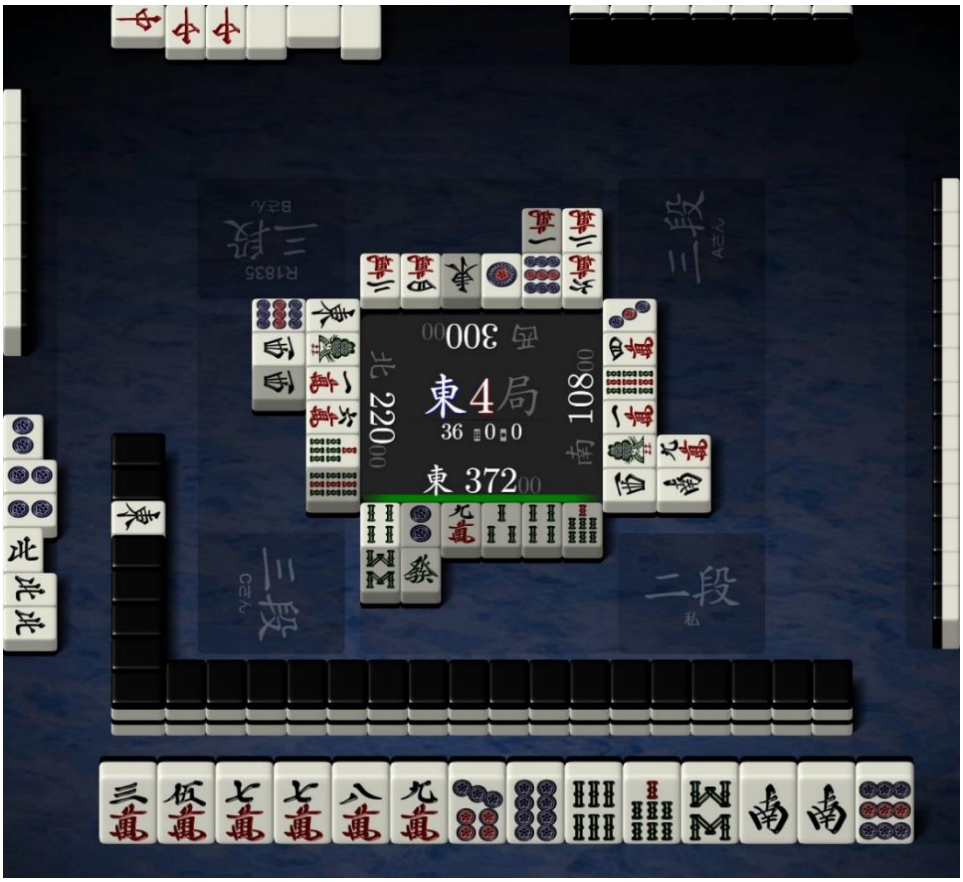


Figure 5. Example screenshot of the Tenhou mahjong client. Retrieved on 9 December 2020.

Just like in the game of chess, riichi mahjong's ruleset is mostly set in stone. The ways in which an individual mahjong video game can differ from the rest are thus limited, and require innovation to attract players, while keeping the base game experience intact. As a general design principle, the basic gameplay features should be

prioritized so that the user experience feels solid and leaves a good impression. For instance, the player interface should be intuitive to avoid unintentional moves when calling or discarding tiles. Graphic elements should be clear so that tiles can be easily distinguished from one another, both in the player's hand and in the discard piles. Interactive things such as menus and tile calling opportunities should provide both visual and auditory feedback to the player. The programming should make the game run efficiently and without any bugs. According to Adams (2010, 252.), in the end, innovation when designing the game should bring out a very small portion of the total fun that the game has to offer, if the game is designed well. The graphics and their clarity, as well as the game's performance can be particularly affected by the game engine—therefore they should also be some of the main points to keep in mind when evaluating Godot Engine's 3D game development capabilities.

4 Godot Engine

4.1.1 Status

Godot Engine is a free open-source game engine created by Argentinian developers Juan Linietsky and Ariel Manzur, written in C++ and released under the MIT license. The engine came to be initially in 2007, and has been in continuous development by its steadily growing community since its first public release in 2014. Its development is mainly funded by subscription-based donations through the membership platform Patreon, as well as sponsoring and occasional monetary grants from companies (Linietsky, 2020). Godot's development environment runs on all the main PC operating systems: Linux, macOS and Windows. Games in it can be built for desktop platforms, the web, as well as mobile platforms including both iOS and Android. The engine does not officially support distribution for consoles like PlayStation and Xbox due to licensing issues with its open-source model, but some third-party console support does exist (Godot Docs – Console support in Godot).

Godot was largely unheard of until more recent years (2019 onwards), when it grew more popular as an engine of choice in the indie game development industry. The

engine has also been praised by numerous companies for the freedom offered by its open source model, as well as its general high quality and usability. (Linietsky, 2019.) It's also well-received for its licensing model that benefits developers in not having to pay subscription fees or royalties for their games, and can be assumed as one of the large factors behind the engine's rapid growth.

4.1.2 Features

Godot features support for both 2D and 3D game development (see Figure 6 and 7), each with their own built-in engine, including VR support. Especially its 2D development capabilities have been praised. Main 2D features include a tile map editor, lighting and normal maps, collision detection, and animation support. The 2D engine also uses pixels as measurement units, which according to Fat Gem Games co-founder Shane Sicienski (as cited in Dealessandri, 2020) makes designing pixel art games much easier, as opposed to an engine like Unity. These can be seen as large contributing factors to the engine's growing popularity, especially since 2D games are still fairly popular in the indie game industry. Godot's 3D engine, on the other hand, supports full MSAA and FXAA anti-aliasing; light scattering effects like reflection, refraction and anisotropy; advanced lighting techniques such as baked global illumination; many mid- and post-processing effects such as depth of field, fog and HDR; and an easy-to-use shader editor. (Godot Engine – Features.) Many of these features, especially anti-aliasing and anisotropic filtering, are important for table games like mahjong, where objects are often viewed from a distance and narrow angle.

Dealessandri also quoted Nathan Lovato—the founder of GDQuest, a YouTube channel featuring mainly Godot-based content—in saying that 3D isn't Godot's strongest side, as it currently lacks in optimization, and loses in performance compared to prevalent engines like Unity and Unreal Engine when it comes to highly detailed game worlds. Brandt (2020) claimed that some contributing factors to its inferior performance are lacking features like proper occlusion culling, limitations on the amount of dynamic lights per object, and weak support of the popular FBX format in asset importation. It's important, however, to consider that Godot is in

active development, and technology in the tech industry advances at a fast rate. Therefore even comments made a year prior to the day of writing this study might not give a completely accurate assessment of the engine's status quo in performance, and should be taken with a grain of salt. For instance, its FBX importer was rewritten for the Godot version 3.3 release on 21 April 2021, making Brandt's claim regarding the feature now obsolete (Verschelde, 2021). Godot also doesn't yet have a large amount of games published under it, which can be theorized as a result of inferiority in terms of available resources and performance, but the suitability of the engine needs to be evaluated on a project by project basis. Over time, new features to the engine will be added, and these statistics will also likely improve.

Figure 6. Example of a 2D game developed in Godot. *Screenshot of Slyway by Guaranapps*. Godot Engine – Features. Retrieved from https://godotengine.org/features

Figure 7. Example of a 3D game developed in Godot. *Screenshot of Infinistate by fracteed*. Godot Engine – Features. Retrieved from https://godotengine.org/features

Major features that both the 2D and 3D engine have in common are the node tree and scene system. Nodes are the building blocks for composing games in Godot. They all have a unique name, adjustable properties, and can be used to display images, 3D models, cameras, sounds, animations, et cetera. Multiple nodes together form a tree structure, where parent nodes can have child nodes, as seen in Figure 8. This allows the developer to organize their projects more efficiently, especially when

combining different nodes to create more complex functions. (Godot Docs – Scenes and nodes.)

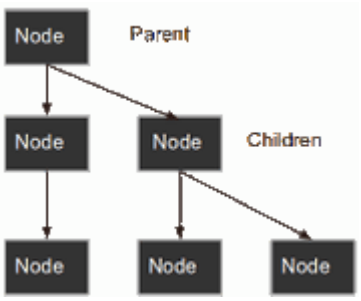


Figure 8. Godot node tree structure. Godot Docs – Scenes and nodes. Retrieved from https://docs.godotengine.org/en/stable/get-ting_started/step_by_step/scenes_and_nodes.html

The node system is closely tied to the scene system. A scene in Godot is essentially a node tree, except it only has one root node, and can be instanced as well as saved on the disk. Instancing scenes is the act of taking an independent scene and including it as a part of the node tree structure of another scene, hence simultaneously making it into a subtree, as seen in Figure 9. For example, a player scene could be instanced into a world scene, or a single ball scene could be instanced multiple times in a game of pool. This aids in organizing projects, making them more flexible and modular, and helps to avoid repetition. (Godot Docs – Instancing.) When running a game in Godot, the engine will play whatever scene is currently marked as the “main scene”, which

essentially is the game itself. Consequently, the Godot editor can simultaneously be thought of as a scene editor. (Godot Docs – Scenes and nodes.)

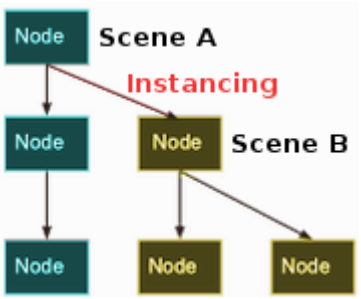


Figure 9. Scene instancing. The node tree Scene B is instanced as a subtree under Scene A's root node. Godot Docs – Instancing. Retrieved from https://docs.godoten-gine.org/en/stable/getting_started/step_by_step/instancing.html

The data that nodes use to display content with is retrieved from resources, which is a specific datatype in Godot used for data storage. The relation between nodes and resources is displayed in Figure 10. Anything saved or loaded from disk is considered a resource. They consist of textures, scenes, scripts, fonts, animations, audio, and translation files, which are used for storing text localizations in multilingual games. Resources are loaded only once, and can be saved either externally or be built into scene files. They're also automatically freed from memory when no longer in use. It's possible to script custom resources in Godot with their own properties and methods, which allows for use cases like enemy stats being stored in one tangible format that can be easily serialized. (Godot Docs – Resources.)

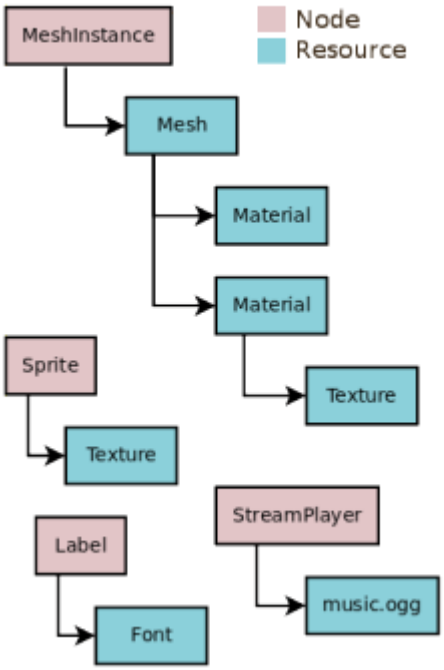


Figure 10. Relation between nodes and resources. Godot Docs – Resources. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/resources.html

4.1.3 Programming

The engine supports a variety of programming languages: C#, C++, VisualScript, and Godot's very own GDScript. The latter two are considered the main languages, and have been integrated into the editor, thus being more accessible to developers. On the other hand, C# and C++ require their own IDE to be used, but can have better performance. (Godot Docs – Scripting.) VisualScript in particular is aimed at people without programming experience, as it relies on graphical means to depict connections between parts of the code, requiring less abstract thinking. It's heavily based on writing functions, which are then visualized with canvases that have nodes connected to them (see Figure 11). These nodes are functionally different from the node trees that work together with scenes. The nodes have ports that can be used to direct data in and out of the function via connections, akin to giving parameters and returning values. A single script can consist of multiple functions, as well as other

nodes like variables, actions, constructors, typecasts, and et cetera. While VisualScript is a great alternative for non-programmers like artists and designers, the downsides usually come with slower development speed and difficulties with making changes to the code. (Godot Docs – Getting started with Visual Scripting.)

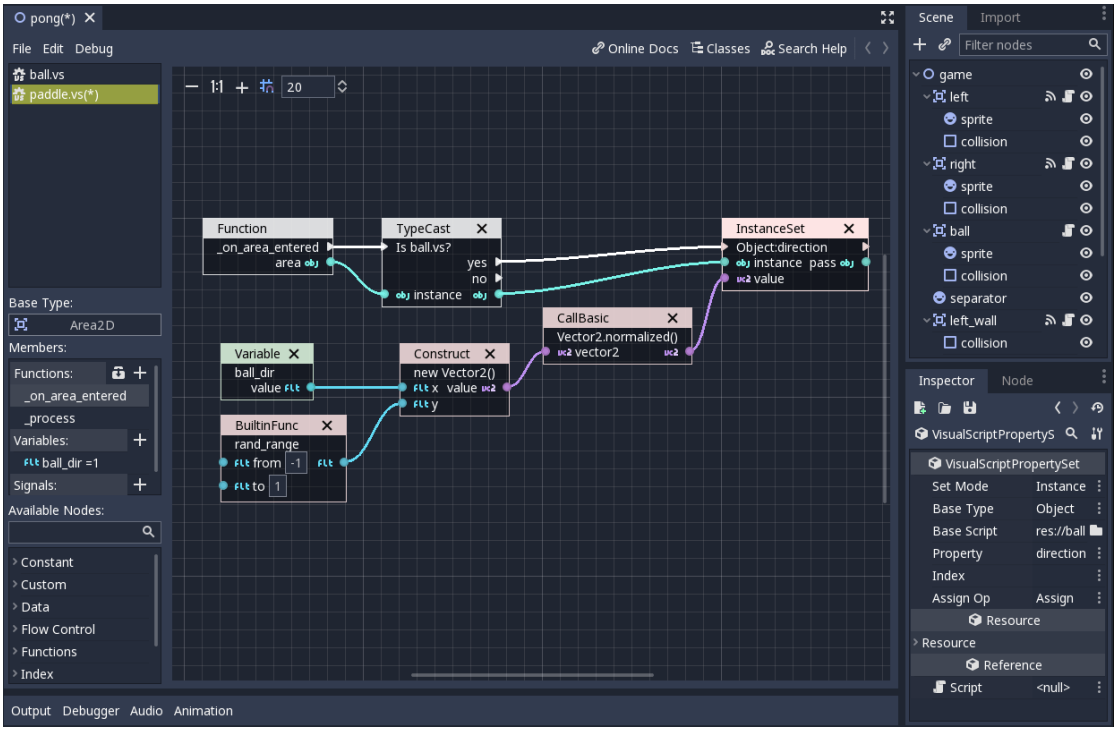


Figure 11. VisualScript editor in Godot. Godot Docs – Getting started with Visual Scripting. Retrieved from https://docs.godotengine.org/en/stable/getting_started/scripting/visual_script/getting_started.html

Being integrated into and optimized for the engine, the language recommended by Godot's documentation, especially for novice programmers, is GDScript. GDScript is a high-level dynamically typed programming language, which uses a syntax resembling Python. Its structure therefore uses indentation to differentiate code blocks, such as if-else statements and functions, making the code less cluttered. Due to being dynamically typed, changes to the code can be made with ease, and running the project is faster than with a compiled language. However, its performance is worse when compared to statically typed languages, and when it comes to computationally

intensive games, a more performant language like C++ can be seen as a better option. (Godot Docs – GDScript: An introduction to dynamic languages.)

Languages can also be mixed together by using GDNative, a module that allows adding “Native” script files to nodes, which in return can be used to load shared libraries and third-party code (Herzog, 2017). For example, in an otherwise GDScript-written game, a script that performs movement logic and collision checks for thousands of in-game objects could instead be ported to run on C++, yielding performance gains. Godot also features an integrated unit testing framework “doctest” for C++, but for other languages the developer will have to rely on third-party tools, of which there are some (Godot Docs – Unit testing). Unit testing is an important aspect to consider when developing a long-term game that relies a lot on the accuracy of different types of calculations, such as in mahjong’s scoring system. However, unit tests were omitted for the development scope of the prototype version of the project.

Godot also has its own implementation of the observer pattern, referred to as signals. The observer pattern is a software design pattern, where a subject object keeps a list of its subscribed dependents (observers), and notifies them of any state changes by calling one of the observers’ methods. This is also known as event handling in most programming languages. Signals allow nodes to send out a message to other nodes that are listening for the signal, and then do something based on it. They can be used for timers (Figure 12) to detect when the timer runs out, without having to actively wait for the timer to count down before executing other instructions. Another common use case is to detect whether an object, such as a button, was clicked. Signals have to be connected from nodes to other nodes either through the Godot editor, or from the code, in order to be emitted.

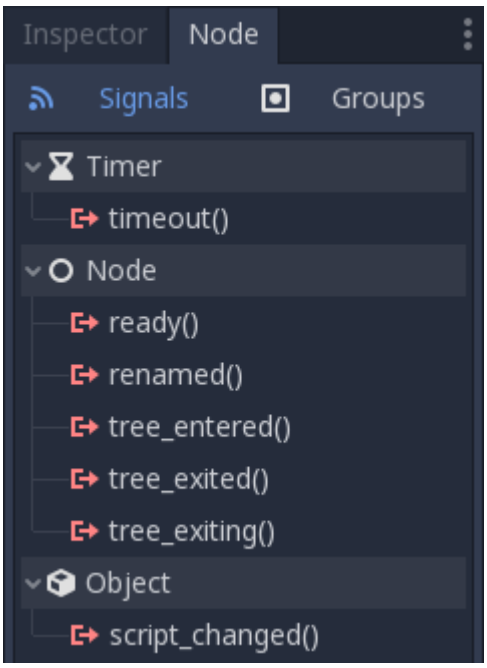


Figure 12. Available signals for a timer node. Godot Docs – Signals. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

4.1.4 Shaders

Shaders are pieces of software that are run on the graphics processing unit (GPU). They're commonly used for post-processing improvements in movies, but are also used to create digital art and animation via computer-generated imagery, also known as CGI. In video games, they are used for creating a wide range of different visual effects, by allowing developers to inject their own code into the GPU graphics pipeline. Effects that are accomplished using shaders include shadows, blur, liquid simulation, dissolving, animation, and glowing. Even though they need to be run millions of times per second, shaders work extremely fast due to parallel processing, which is provided by the GPU architecture. In order for this architecture to be taken advantage of, application programming interfaces (APIs) are created and managed by groups and companies like Microsoft and Khronos Group, which provide functions for game engine developers to use in their graphics rendering engines. Some popular graphics APIs include Microsoft's DirectX, as well as Khronos Group's established cross-platform OpenGL and the emerging Vulkan API. Shaders are then written by

developers using special shading languages, such as OpenGL's GLSL ES 3.0, with an interface provided by the game engine. (Hasu, 2018.)

Godot's graphics engine runs on the OpenGL API, with two available rendering backends: GLES2 and GLES3, which respectively map to OpenGL 2.1 and 3.3 on the desktop, ES 2.0 and ES 3.0 on the mobile, as well as WebGL 1.0 and 2.0 on the web (Godot Docs – Differences between GLES2 and GLES3). There are three main types of shaders available in Godot: spatial (3D), canvas_item (2D) and particle shaders. A render mode can also be specified, if the intention is, for example, to disable object culling or to leave the object unshaded. Furthermore, spatial and canvas_item shaders may be overridden using vertex and fragment processor functions. Vertex processors affect object vertices, and can be used to dynamically manipulate the object's position, rotation, and scaling, as well as its UV coordinates for texture mapping. Fragment processors, on the other hand, are run for every pixel in the object, and are generally used to set up material properties like roughness, albedo color and rim. Godot's shading language is highly similar to GLSL, with added functionality but a bit less flexibility. (Godot Docs – Shaders.)

It's possible to make both regular code-based, and visual node-based shaders in Godot. Node-based shaders are called VisualShaders (Figure 13), which bear similarities to the aforementioned VisualScript in terms of workflow. They're also more accessible to artists than writing code-based shaders, as using them doesn't require any programming knowledge. In a similar fashion to VisualScript, VisualShaders are created by connecting input and output nodes together. The input data is usually manipulated by some function nodes, such as the Fresnel function node in Figure 13. VisualShaders are converted into code-based shaders behind the scenes, but it's also possible to manually convert them into code format at any given time. Not every effect is possible to make using VisualShaders, so a combination of them and code-based shaders might sometimes be necessary in order to get the desired effect to work. (Godot Docs – VisualShaders.)

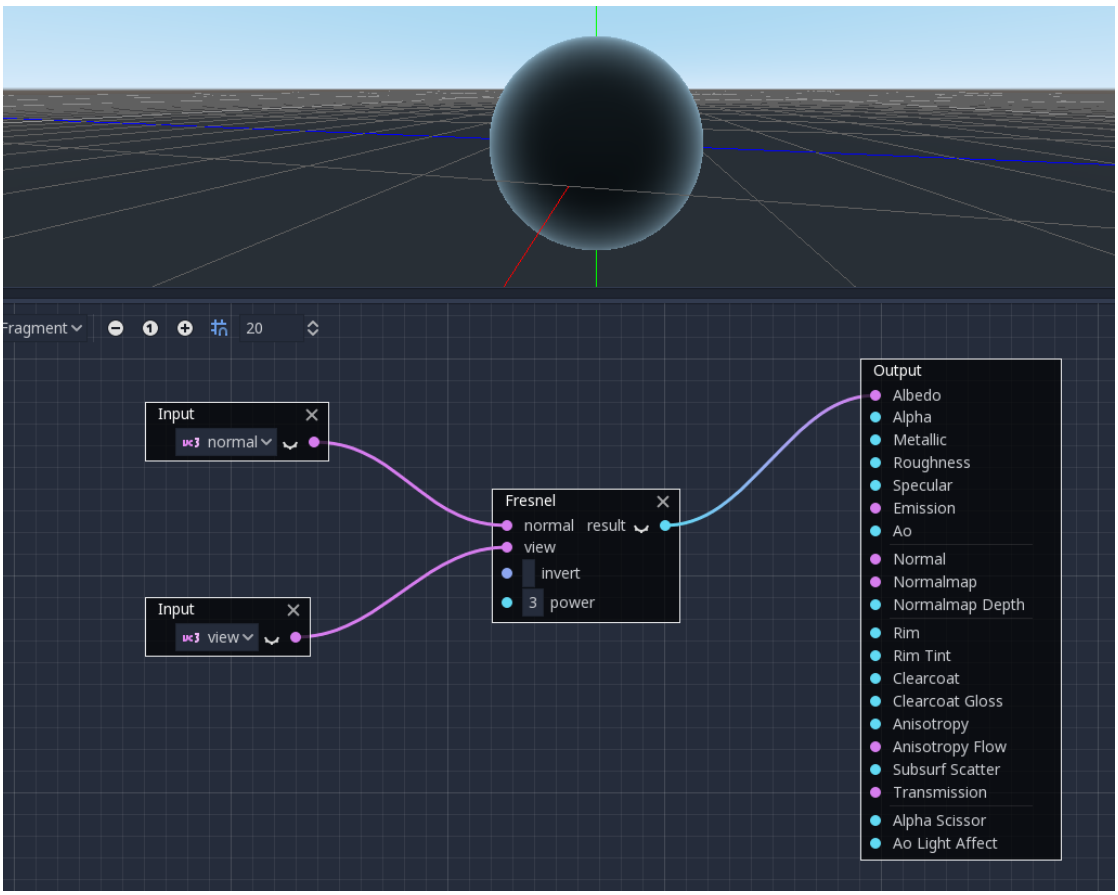


Figure 13. VisualShader creation in Godot. Godot Docs – Getting started with Visual Scripting. Retrieved from https://docs.godotengine.org/en/stable/tutorials/shading/visual_shaders.html

4.1.5 Available resources

The engine's official documentation has been localized to multiple different languages, and Miettinen (2019, 8) found that it's also easily legible. Godot's community is spread around multiple social media sites like Facebook and Reddit, and is also very active on the messaging platform Discord. The engine's main website also sports a community Q&A section, where questions and answers can be rated with an upvote system. (Godot Engine – Community.) In addition, as the engine is open-source, it has an active GitHub page where issues like bug reports can be posted, and where any person can contribute to the engine's development. In practice, many of the posted issues also act as a form of technical support, with the community occasionally offering workarounds to problems encountered during

development. As of the writing of this study, the engine's repository has over 1,400 contributors, with over 1,100 pull requests and over 34,000 total commits. (GitHub.) This gives an idea of the size of the community that's involved in the engine's development. The engine's development is also prioritized based on user feedback and proposals, to determine what features are worked on next (Nations, 2021).

However, according to Godot contributor and Little Red Dog Games' project director Ryan Hewer (as cited in Dealessandri, 2020), despite Godot having a tight-knit and supportive community, its size is still meager in comparison to one like Unity's. He continued by stating that certain resources like tutorials and asset packs can be found lacking when compared to Unity. Despite this, Godot does have its own, albeit small, asset library, mostly featuring (depending on the license) free user-submitted development tools and game demos (Godot Docs – About the Asset Library). Certain third-party asset stores also exist, such as Godot Marketplace, where community members can buy and sell content such as models, sound packs and graphical interfaces (Godot Assets Marketplace). According to Linietsky (2021), a new version of the asset library is in development, which would allow buying and selling assets, with a share of asset sales going towards Godot's development. Since there weren't yet any mahjong assets available in Godot's official library, original assets were created for the project.

5 Development process

This chapter focuses on the development process of the mahjong game in Godot. Godot was chosen as the subject of the study due to its rapidly growing popularity but relatively unexplored status in the game industry. The main problem that the process seeks to resolve is the uncertainty of how capable Godot is in managing 3D game development. The method of mahjong as a case study was chosen for its potential to be represented in a 3D game world and for its multiplayer applications, but also for its manageable size as a one-man passion project. Each part of the process related to Godot was detailed here, and my personal experiences were documented as well. The programming aspect of the game is discussed shortly in the

results, but was not included in the development process, as it's not strictly related to the game engine itself. The versions of Godot used during development were mostly 3.2.3, and later 3.3 and 3.3.1. The chosen programming language was C#, as its use in Godot is less explored compared to GDScript.

5.1 Project setup

When opening up Godot, I was greeted with the project manager window, as seen in Figure 14. Here it's possible to either create a new project from scratch or to use one of the template projects provided by the community, based on the content from the Godot asset library. When creating a new project, the user is prompted for a project name, directory path, and most importantly the renderer to be used. It's possible to choose between OpenGL ES 2.0 and 3.0, the former being a more compatible option especially for web games and older hardware. 3.0 is recommended for better visual quality, hence it was chosen for this project. The renderer can be changed later if it turns out that it's more feasible to make the game web-based instead, or if the better renderer causes performance issues.

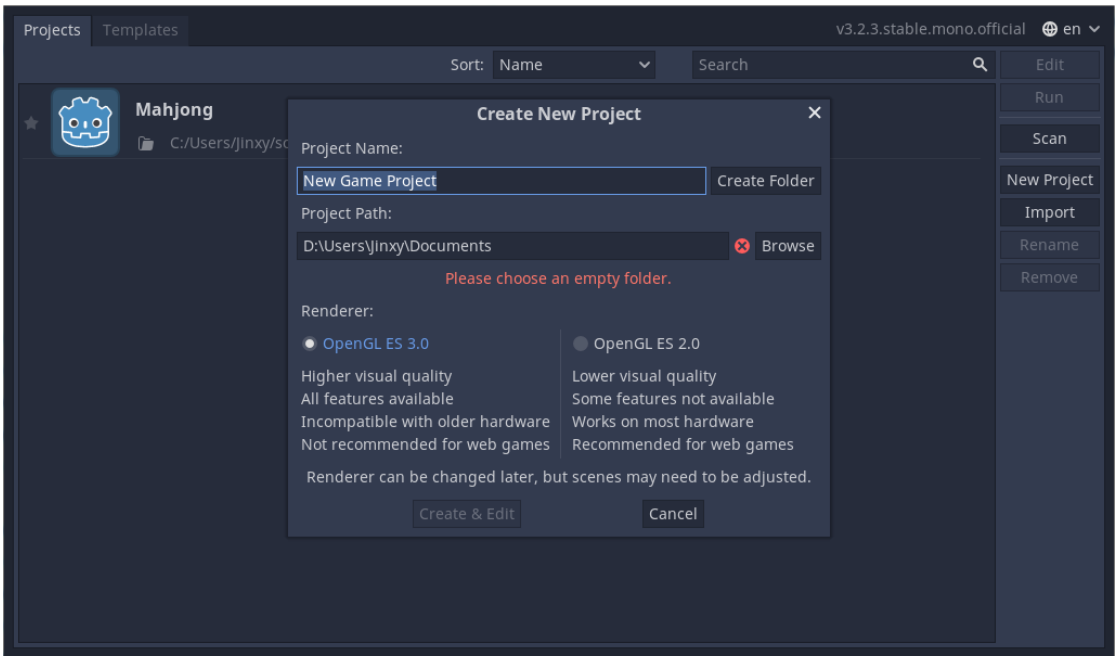


Figure 14. Godot project manager window. Retrieved on 16 March 2021.

After creating a new project and choosing to edit it, a new window opened, presenting the Godot scene editor, as seen in Figure 15. This window allows the user to create, edit, and manage scenes together with their nodes. It's also possible to switch between 2D and 3D perspective at will, as well as access the integrated script editor, by switching to the respective mode on the workspace bar (blue). It's good to note that the project isn't bound to being strictly 2D or 3D, but allows the user to flexibly add and remove both kinds of scenes from the left-side scene dock (orange). Below the scene dock is the file system dock (yellow), which displays the project resources located in its folder. Resource files, such as images, can be dragged into it to be included in the project, in which case their import-related settings (such as texture settings) can be changed from the scene dock's import tab. On the right side is the inspector dock (green), where the main properties of resource files, such as the materials of imported scenes, can be tweaked. The top-right bar allows changing the renderer, as well as the project to be run and stopped at will. The center window (red), also known as the viewport, is used for navigating the active scene by adjusting the camera position and angle, and for editing the scene with the translation, rotation and scaling tools. The bottom panel (purple) lets the user see the console output, debug information, as well as tweak audio settings and animation keyframes.

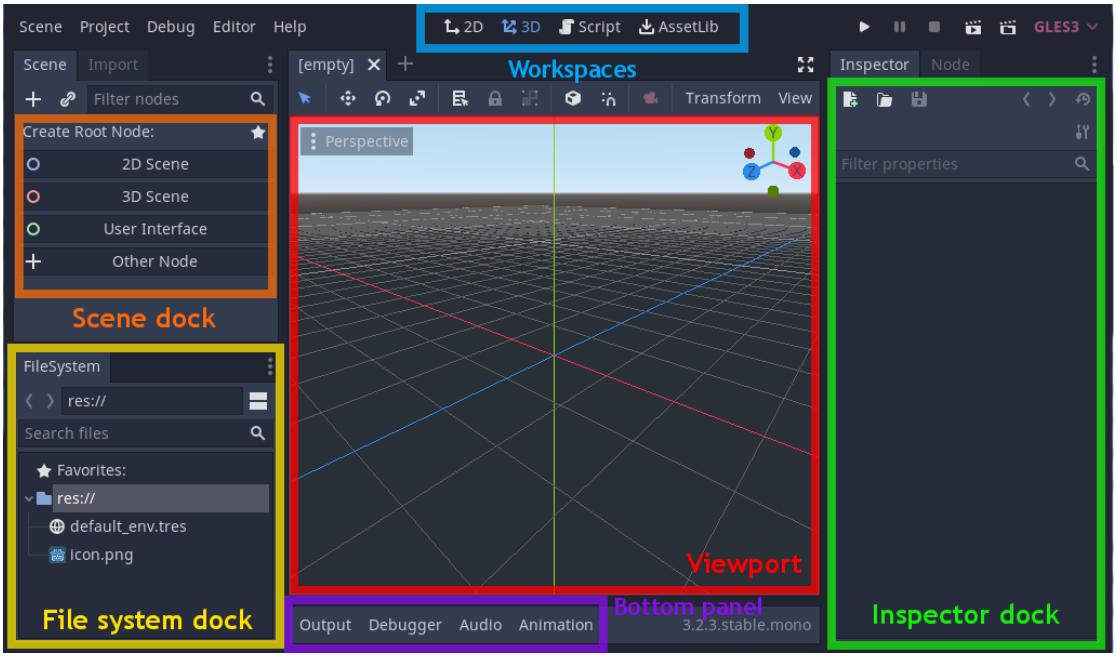


Figure 15. Main view of the Godot editor. Retrieved on 16 March 2021.

Lastly, version control was set up. Version control provides more stability to the project's development cycle. If at any point the project were to be continued in the future, and additional developers were to join in, managing the development in a team will prove to be much easier with version control enabled. Godot has an official plugin for Git version control, although it's not available in the asset library (Godot Docs – Version Control Systems). Integrated Git support may especially be useful for beginner developers that are unfamiliar with the Git command line interface, but want to rapidly contribute to a team project. After downloading the plugin from its GitHub page, the contents were extracted into the Godot project folder. Then the project was opened through the project manager, and the version control was integrated into the editor from the "Project" file menu by selecting "Version Control" and under it "Set Up Version Control". It's also possible to shut down version control from the same menu if needed. This created a new tab "Commit" next to the Inspector and Node tabs on the right-side panel, which is a staging area that's directly integrated into the editor, as seen in Figure 16. After setting up version control, it's also possible to commit changes, and view the current status and file changes directly from the editor. However, since connecting to remote repositories and pushing must still be done outside the plugin, command line Git was mostly used in the development of the project.

Figure 16. Staging area of integrated Git version control plugin in Godot. Retrieved on 26 March 2021.

5.2 Importing assets

Assets in Godot are imported by moving them into the project folder, or alternatively by dragging them into the file system panel. Imported assets are turned into resources, which can then be used in the editor together with other types of resources such as materials. When renaming or moving resources, it's important to do it in Godot's file system manager, as doing so will update all their references correctly. Assets are also automatically reimported if an MD5 checksum difference is detected in the source file. When importing files, Godot created .import files for certain assets like textures, which were used for storing specific import settings. Some of the import settings needed to be configured in order to display correctly, which was done through the scene dock's import tab. Reimporting large textures,

such as the mahjong mat, took a long time, so changes to import settings should ideally be made in one go.

The engine supports many file formats for scenes and models created in 3D modeling software such as Autodesk Maya and Blender, which are also imported as scenes. The recommended scene format is glTF 2.0, specifically the .glb binary format, which is fast to parse and fully supported. Historically, COLLADA has also been a popular format option. However, using it is discouraged as its default exporters are often buggy, and the resulting file is usually larger in size and slower to read compared to glTF 2.0. (Linietsky 2017; Godot Docs – Importing scenes.) A mahjong tile model (Figure 17) was created in Blender, and exported in glTF 2.0 format. The model uses three materials, one for the bottom of the tile (orange), the sides of the tile (white), and the face of the tile, which is also white but uses a separate material for the purpose of texturing. It's noteworthy to mention that unneeded objects like lights and cameras should be removed or tagged with the "-noimp" flag before exporting, as otherwise they will be included in the imported scene. By default, when importing the scene, Godot splits materials in their separate files instead of embedding them, so that they may be edited. This behaviour can be changed from the preset menu in the import tab, but in this case the intention was to edit one of the materials later.

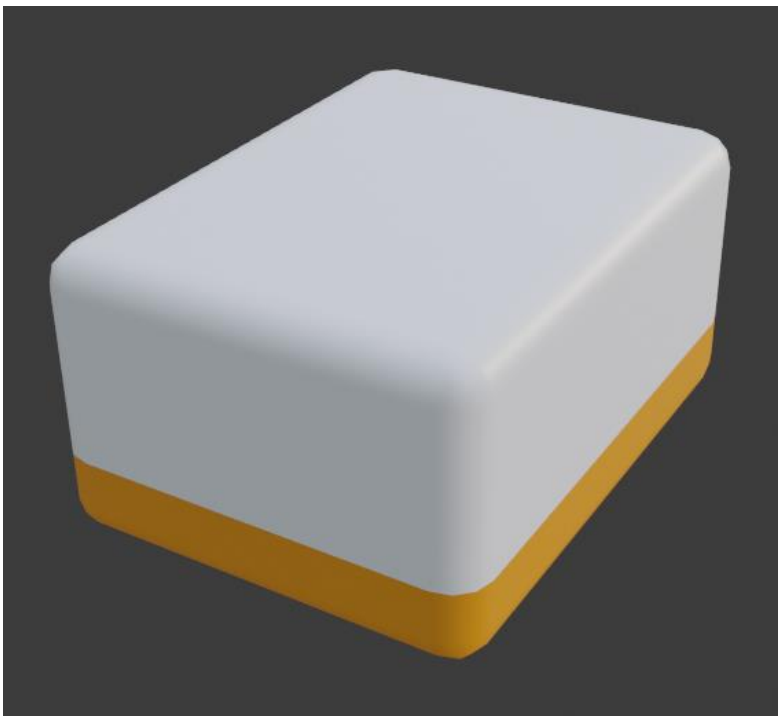


Figure 17. Mahjong tile model created in Blender. Blender editor. Retrieved on 27 April 2021.

A set of slightly edited third-party tile textures was compiled into a texture atlas, as seen in Figure 18. From top to bottom, the suits in the atlas are dots, bamboo, characters, and finally the wind and dragon honor tiles. The blank space in the figure between the red and green dragon is the white dragon. The atlas was used in-game to render the tile faces onto the mahjong tiles by changing UV coordinates for each individual tile mesh. Hence, ultimately only one tile model and texture were needed to display all 34 kinds of them. This technique, also known as batching, effectively reduces draw calls, improving game performance especially if a mobile version were later to be developed. A normal map was also baked using a grayscaled version of the sprite sheet, which adds a realistic sense of engraving depth to the rendered tile. When importing the texture atlas and normal map as textures in Godot, certain import settings like anisotropy and filter had to be turned on, which affected their graphical fidelity. In addition, a selection of open source fonts were imported into the Godot project for text styling purposes.

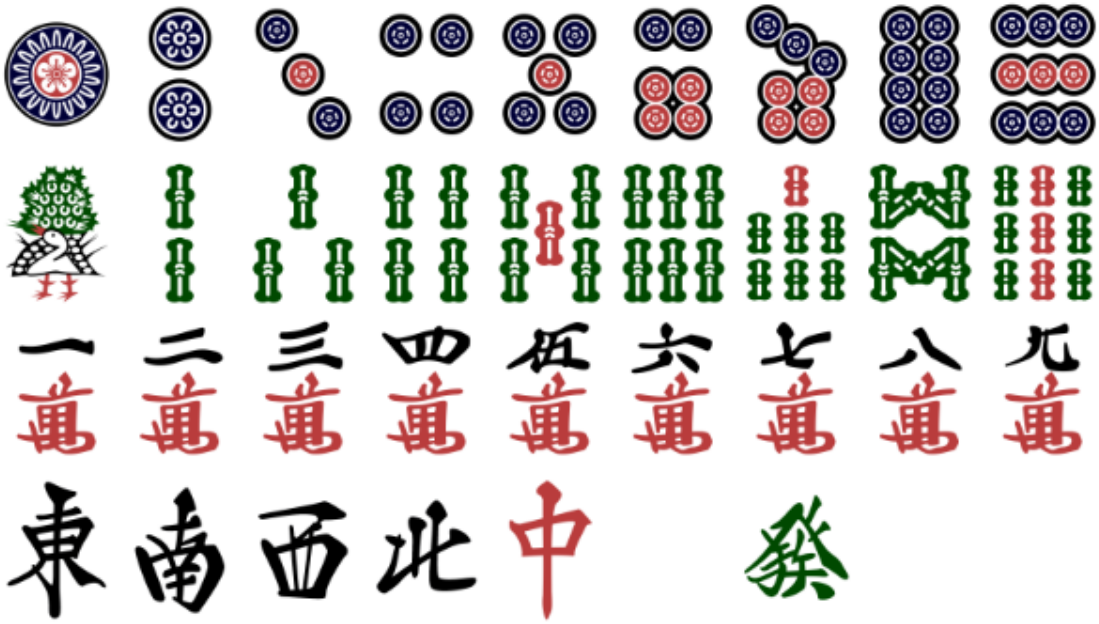


Figure 18. Texture atlas of riichi mahjong tile textures. Retrieved on 1 April 2021. Adapted from “Vector graphics of riichi mahjong tiles”, by GitHub user FluffyStuff, list of .png image files in GitHub repository (https://github.com/FluffyStuff/riichi-mahjong-tiles/tree/master/Export/Regular). CC BY 4.0.

5.3 Creating nodes and scenes

Every project in Godot starts by adding a root node from the scene dock, as seen in orange in Figure 15. Because the intention was to make a 3D game, the 3D scene spatial node was chosen. A three-dimensional bounding box, also known as a gizmo, was then appended to the center of the viewport, as seen in Figure 19. Gizmos are tools used for manipulating objects in three-dimensional space, and can be used to translate, rotate and scale objects at will. The first goal was to create a table scene, which is the standard platform that real-life mahjong is played on. Automatic mahjong tables generally have borders, which can have a texture resembling wood. The borders themselves are usually used by the players for stacking tiles against it when sorting their hands. All scenes in Godot are constructed by adding nodes together to form a tree. Geometrically, a table can then be formed of a flat plane as the mat, as well as four scaled cubes, which are the borders. Five MeshInstance nodes were

therefore added to the scene as the building blocks for the table, as seen in Figure 20.

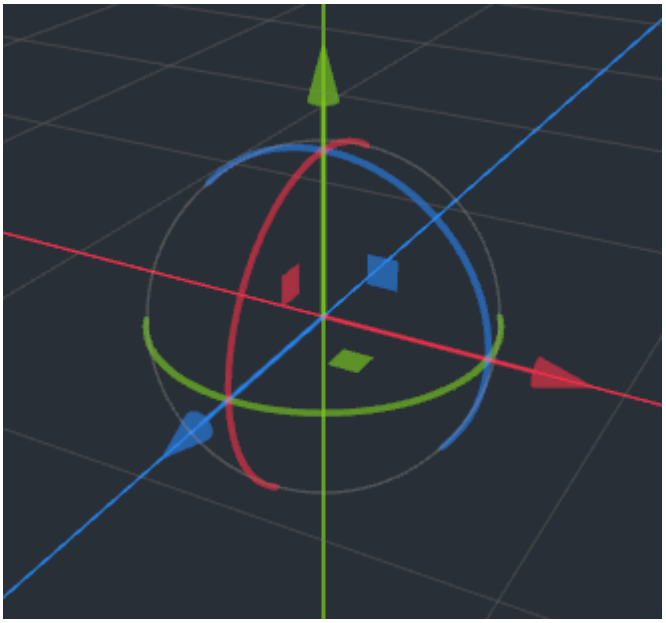


Figure 19. Gizmo in the Godot 3D editor. Arrows are for translation, rings for rotation, and squares for scaling. Retrieved on 22 May 2021.

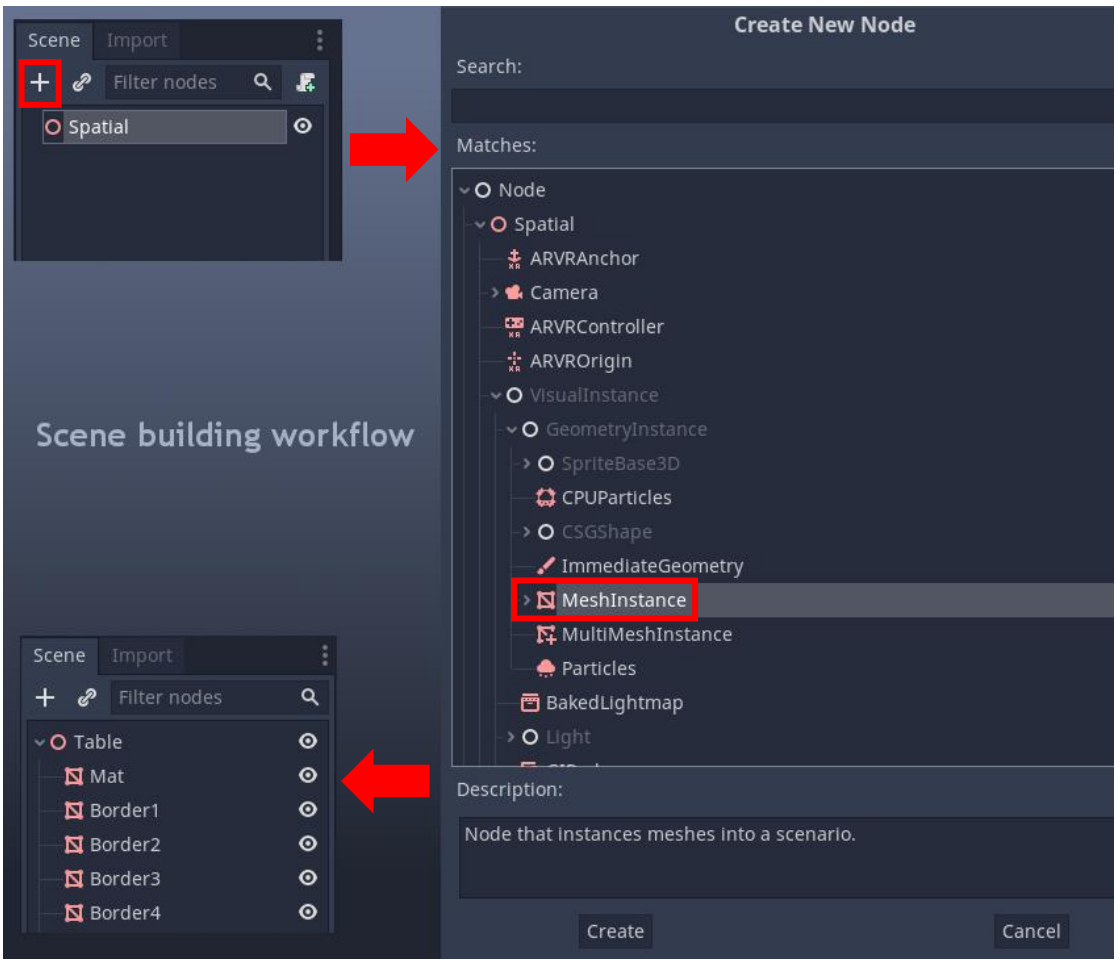


Figure 20. Process of adding nodes to scenes in Godot. Retrieved on 22 May 2021.

One of the MeshInstances was then defined as a plane mesh, while the other four were defined as cube meshes, and were then scaled to fit around the table plane's borders. It was possible to configure snap settings for both translation, rotation and scaling, which allowed for more precise positioning to be done through the gizmo. A table mat texture was supplied by an artist who was working together on the project with me, and then applied to the plane, as seen in Figure 21. A new material with a transparent wood texture was also created for the borders. The table scene was then later instanced into the main game scene as its own node, as showcased in Figure 22. There were multiple benefits of instancing scenes like this, including improved modularity, as the scenes could still be edited on their own, and the saved changes were then updated to all instances of it in other scenes. Later on, other nodes were added to the tile scene, which is especially when the modularity benefits came into use.

Performance was maintained even when multiple clones of the same object needed to be instanced at the same time. Such is often the case with game objects like city buildings and collectable items, and in this case, mahjong tiles. This can be seen in Figure 22, where multiple tile scenes were instanced into a player hand scene, and multiple player hands were then instanced into the main game scene, to represent the player's opponents. An instanced scene's children nodes can't be edited in the parent scene by default, but it is an option. For this purpose, the scene has to be "made local", after which any changes done in the original scene aren't reflected onto the local scene anymore, effectively making it unique from its clones. Without prior knowledge in Godot, the workflow of building the game world with scenes was also very intuitive and easy to learn.

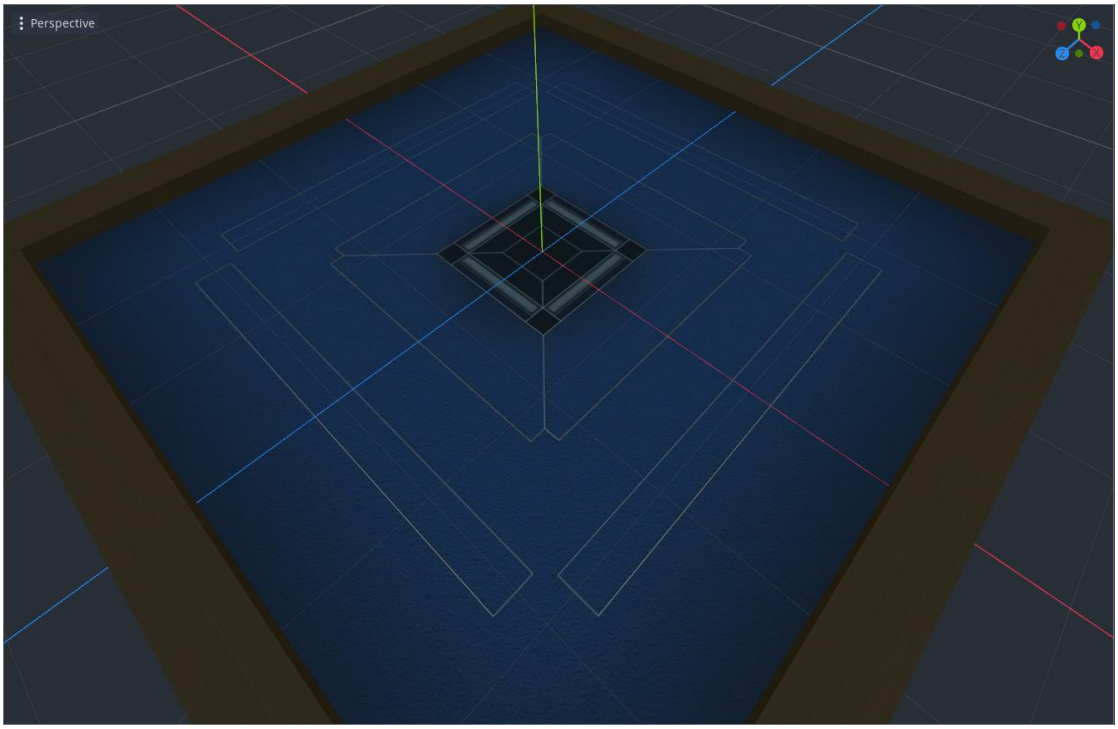


Figure 21. Mahjong table scene created in Godot. Retrieved on 22 May 2021.

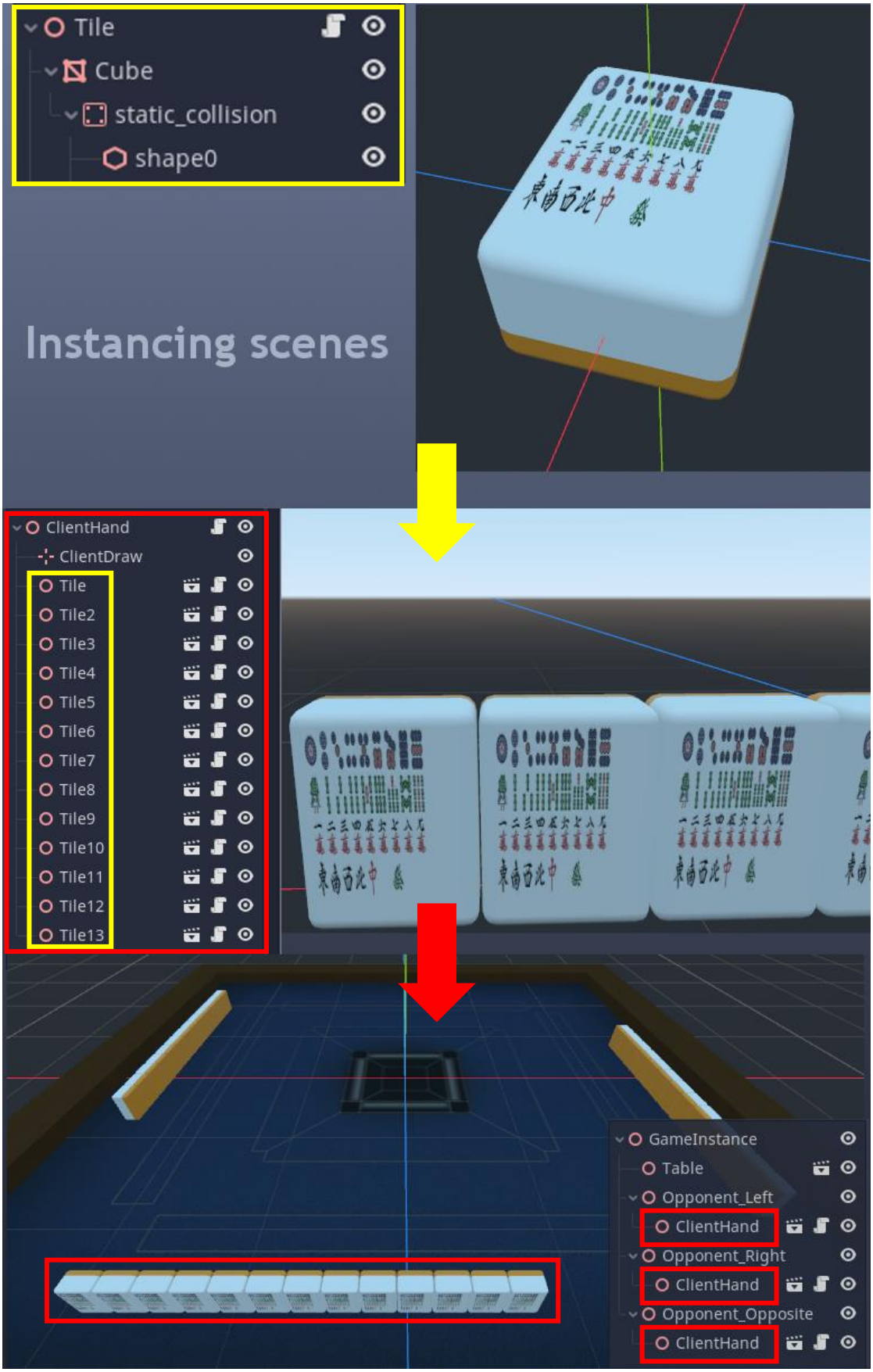


Figure 22. Instancing scenes in Godot. Retrieved on 22 May 2021.

5.4 Designing the user interface

Nearly every game comes equipped with a graphical user interface (GUI) nowadays. GUI components include standard navigation elements such as menus and buttons, but also heads-up display (HUD) elements like health bars, minimaps, and object text labels, which are specific to video games. User interfaces in games can be further divided into groups like diegetic and non-diegetic components, depending on whether they're a part of the game's story and/or space. Non-diegetic components in mahjong would be the player's turn timer and call buttons, which should pop up whenever the player is able to call a tile. They're not diegetic, as no character in the game sees them aside from the player themselves. On the other hand, diegetic ones are all the information that's displayed in the center of the screen, namely the amount of points left per player, or remaining tiles in the wall, which every player should be able to see. Both components are important for the game's usability, so they had to be designed in an intuitive way for the player to make use of them.

General problems with UI design include implementing proper scalability and dynamicity of the interface, in other words the ability to adapt to changes in properties like window size or displayed content. Fortunately, Godot was able to offer many flexible solutions to these problems. A basic main menu (Figure 23) was designed using Godot's control nodes. Control nodes are specifically used to design user interfaces, as they display on top of other nodes, and work with both 2D and 3D scenes. There were a vast amount of different control nodes to choose from, ranging from multiple types of containers to buttons and labels. The containers were especially useful during development of the UI elements, as they provide uniform spacing and alignment to all their child nodes, as seen in Figure 23. This also allowed them to scale efficiently whenever new buttons were added on the fly, or if the game window was resized.

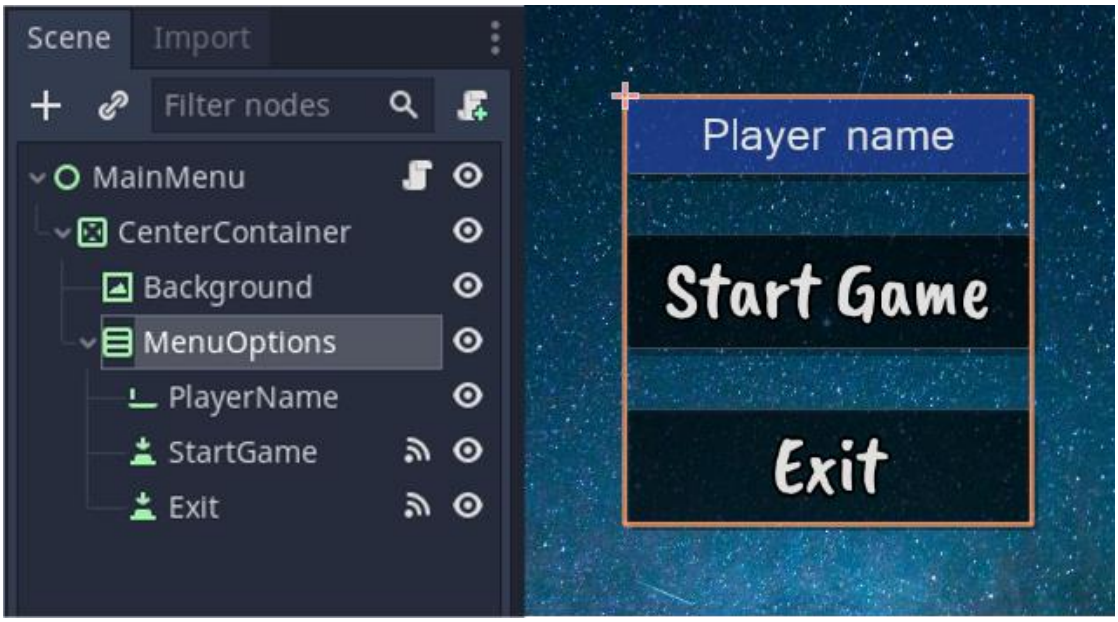


Figure 23. Basic main menu designed with Godot's control nodes. Retrieved on 23 May 2021.

The materials for the buttons and fonts were designed in Godot itself. There were plenty of styling options available for both, including size, opacity, outlines, and shadow effects, many of which can be seen in Figure 24. It was very convenient to be able to style all the UI elements without having to do it separately in an image editing program. It was especially essential for the fonts, as text inside the labels would often change. Without dynamic font styling, the text would have to be styled in an image editing program every time its content changes, which would be an arduous process to repeat. In addition to the main menu, other non-diegetic components like the score screen, call buttons and turn timer were also implemented, which can be seen in Figure 25. These mostly consisted of labels, the values of which were displayed and updated through code whenever needed. The turn timer had to be implemented using a monospace font, as changes in digit spacing would otherwise move the timer around every time it decremented. Signals were also added to the buttons, so that their click actions could be handled in code as well.

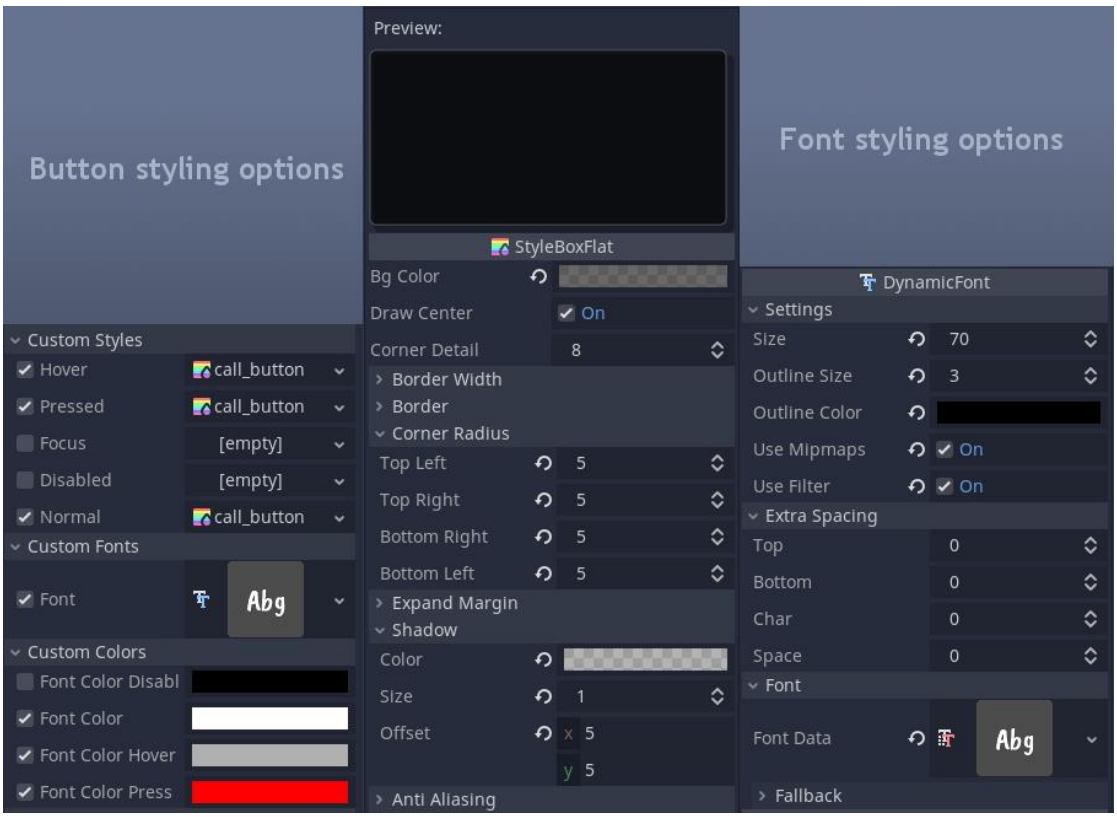


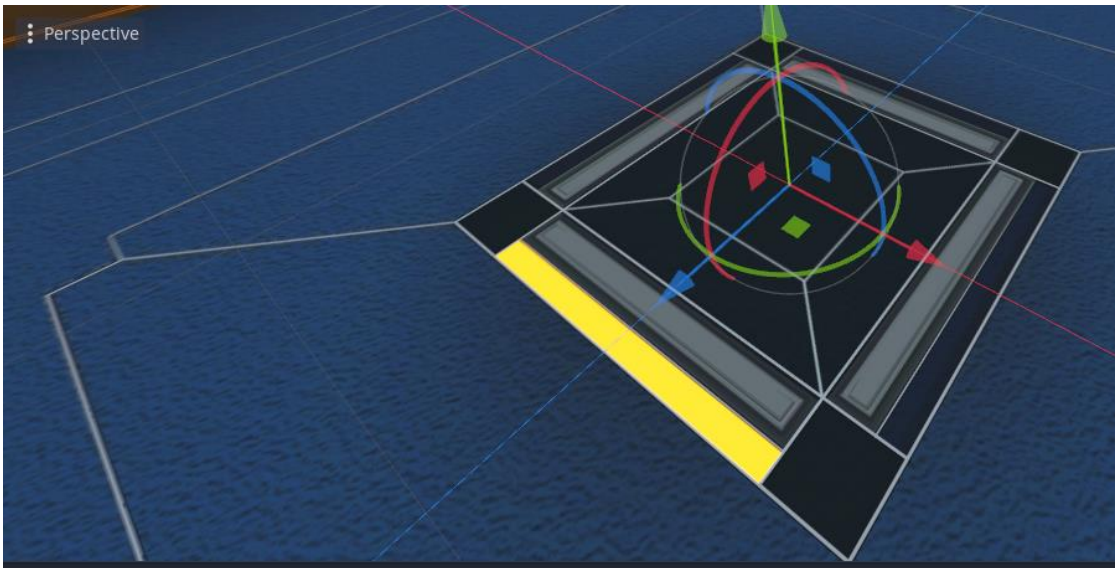
Figure 24. Many of the available UI styling options in Godot. Retrieved on 23 May 2021.

Figure 25. Mahjong score screen (top), and call options bar with turn timer (bottom) designed with Godot control nodes and UI styling. Retrieved on 23 May 2021.

5.5 Writing shaders

Since the project was done in 3D, Godot's canvas_item shaders weren't of importance. No particle effects were implemented either due to lack of time, but a couple of spatial shaders were written for the project. The first one was a fragment shader created for the table mat texture in order to indicate the current player turn with a flashing indicator light. The initial plan was to create 3D sprites which would

be positioned on every player's turn indicator and be made to glow, but it ended up creating anti-aliasing problems with jagged sprite edges when viewed from afar. Instead, an emission texture was created and applied to the table plane, where white parts of the texture were made to glow while leaving alpha values unaffected. This allowed for pixel-perfect implementation of the effect, providing a smoother end result, as seen in Figure 26. The integer "player_indicated" in the shader code is a uniform value, meaning that it can be defined from outside the shader, which made it easy to change the indicated player directly from the game's code. Godot's shading language was easy to work with after understanding the basics of fragment shaders.



Search Edit Go To Help

```
24
25  uniform int player_indicated = 1;
26  void fragment() {
27      vec2 base_uv = UV;
28      vec4 albedo_tex = texture(texture_albedo,base_uv);
29      ALBEDO = albedo.rgb * albedo_tex.rgb;
30      METALLIC = metallic;
31      ROUGHNESS = roughness;
32      SPECULAR = specular;
33      if (player_indicated == 1 && UV.y > 0.58593f || player_indicated == 2 && UV.x > 0.58593f ||
34          player_indicated == 3 && UV.y < 0.41015f || player_indicated == 4 && UV.x < 0.41015f) {
35          vec3 emission_tex = texture(texture_emission,base_uv).rgb;
36          EMISSION = (emission.rgb*emission_tex)*emission_energy * ((abs(sin(TIME * 3f))) + 0.2f);
37      }
38  }
39
```

Figure 26. Mahjong player turn indicated using a fragment shader written in Godot. Retrieved on 23 May 2021.

The second shader created for the project was a VisualShader, for the purpose of off-setting the tile objects' UV values, which directly affect what parts of the texture are supposed to be displayed on the tile face. Modifications with the VisualShader were mainly done to the vertex processor, as seen in Figure 27. ScalarUniform variables "Offset_X" and "Offset_Y" were supplied from outside the shader, which corresponded to the indices of the tile's texture in the tile texture atlas (Figure 18). The tile texture was successfully rendered on the tile object's face based on the two offset values. One problem that followed was that changes to one tile's texture offset values affected all of the tiles at the same time, so every tile would undesirably have the same face. This was solved by enabling the "Local to Scene" option (highlighted in Figure 27) in the mesh settings through the inspector dock, making material changes unique for each tile.

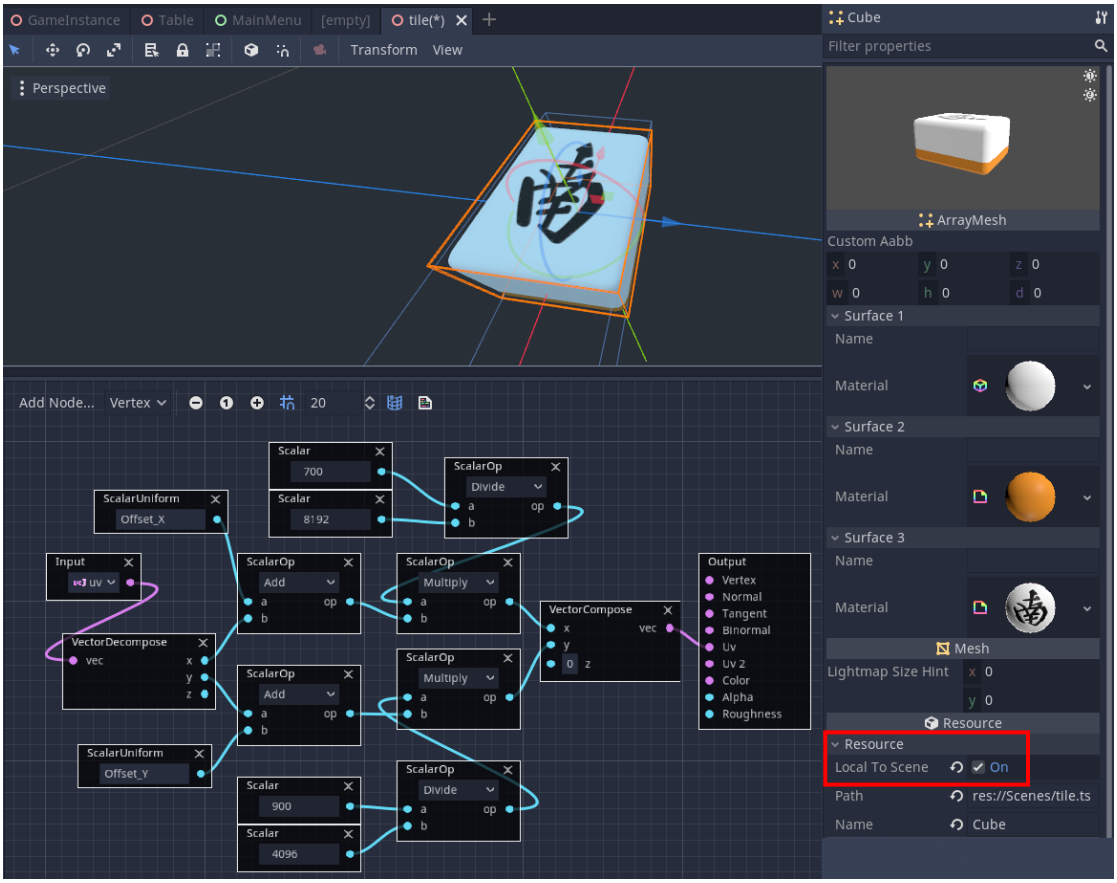


Figure 27. Mahjong tile face displayed by using a visual vertex shader created in Godot. Retrieved on 23 May 2021.

The VisualShader workflow took some time to get used to, but learning it can be a good method of teaching less programming oriented people how shaders work. For larger shaders, it's hard to recommend using VisualShaders due to the extreme size they take on the screen. The more complex the shader gets, the less legible it becomes due to the node-based structure. One positive surprise with both types of shaders was how changes made to the shader code were updated instantaneously in the live preview. Weaker game engines don't necessarily have a live preview at all, in which case the shader code must be compiled every time in order to see the effect. Godot's live preview sped the trial-and-error based process of shader coding up tremendously.

5.6 Setting up cameras, viewports and lighting

A player view consisting of two camera perspectives was implemented using Godot's camera and viewport nodes. There were three camera projection modes to choose from—orthogonal, perspective and frustrum. Of these, one perspective camera, and one orthogonal camera were used. The perspective camera was used as the main in-game camera to display the entire mahjong table, and was positioned far away from the origin with a low field of view of 15 degrees. Meanwhile, the orthogonal camera was used to display the player's hand evenly across the screen. The main viewport could be split into compartments, and each compartment could be given one of the cameras to display as a preview. Being able to see the table from the player's perspective while setting tile positions for the discard pile was quite helpful during development, which can be seen being done in Figure 28.

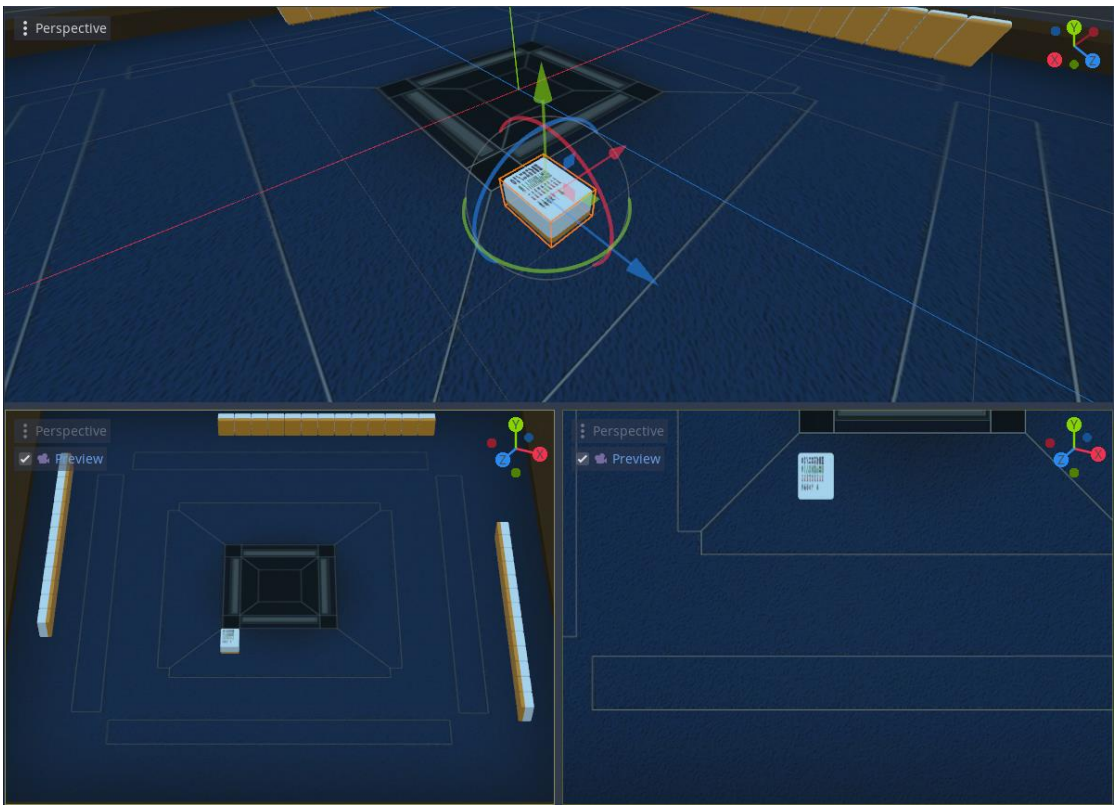


Figure 28. Positioning a tile object with the help of multiple cameras in different viewport compartments. Retrieved on 23 May 2021.

In order to combine multiple camera views, the cameras would have to be put into their own viewports. In this context, viewports refer to viewport nodes, one of them which is the root viewport that cannot directly be seen in the node tree. The player's main perspective camera could thus be put anywhere in the node tree, and it would work. However, for the orthogonal camera, another viewport had to be created. Viewports are essentially surfaces that the game is projected on, and multiple viewports can be visually combined into one using viewport container nodes. This way, they can be used to render 3D graphics in a 2D game, or 2D graphics in a 3D game. In the project's use case, the orthogonal viewport's screen (player hand) was overlaid transparently on top of the main screen. It was possible to assign the viewports into their own "worlds", which meant that Viewport A could be made invisible from Viewport B's projection, but that Viewport B could still be seen from Viewport A. This is demonstrated in Figure 29.

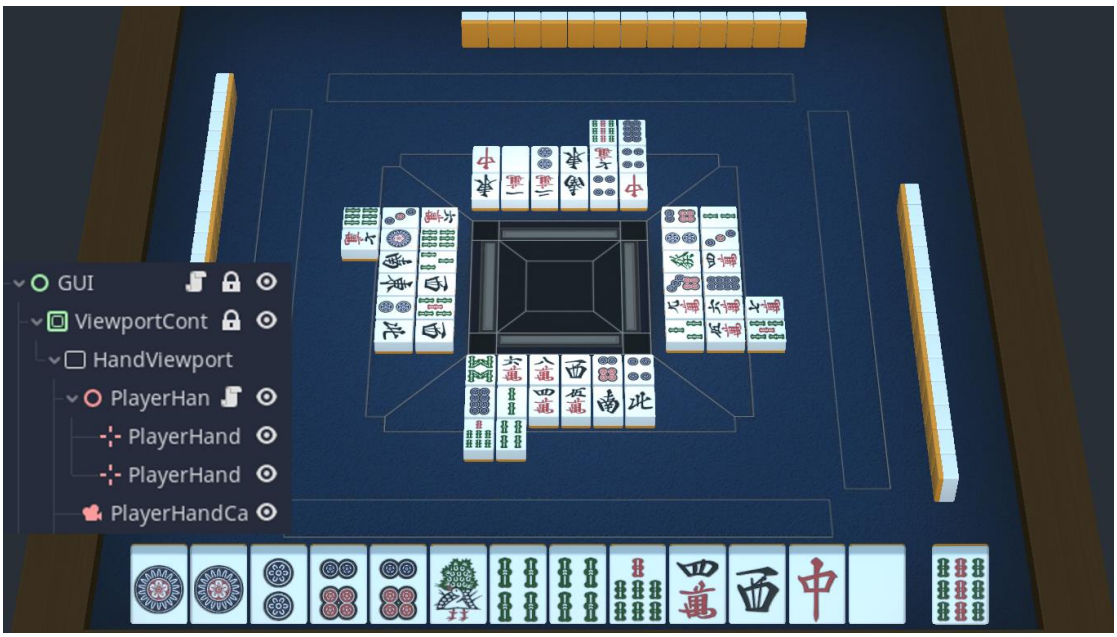


Figure 29. Two viewports combined to overlay the player's hand orthogonally to the player. Retrieved on 24 May 2021.

However, this presented a new problem. When mousing over the tiles in the player's hand, it would be ideal if they were to be highlighted. It is also necessary to be able to register mouse click events, so that the player can choose which tile to discard on their turn. Normally, Godot offers signals that can be set for objects, which are emitted when the mouse is detected to be hovering over the object or clicking it. The details of the clicked object could then be sent via the signal to a signal handler so that the correct tile is discarded from the player's hand. Unfortunately, the ViewportContainer node, which is necessary for getting the other viewport's projection to display, blocked these mouse inputs from being detected. Therefore, a workaround had to be made, where an InputHandler class would then take the mouse position coordinates and cast a ray from the other viewport's camera at those coordinates, checking whether a tile was hit. This method is known as ray casting, and is used for purposes such as AI detection algorithms, and in this case, object picking. Godot's documentation was particularly helpful in solving this problem, with ready-made code examples of ray casting being available. Working results of ray casting can be seen in Figure 30.

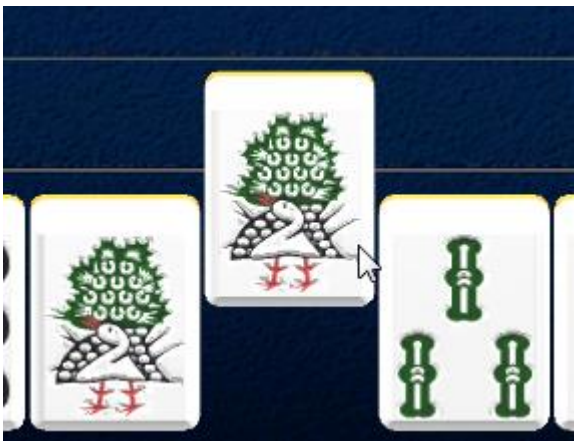


Figure 30. Ray casting used to highlight tiles upon hovering over them. Retrieved on 24 May 2021.

Another use of viewports is in the creation of viewport textures. The diegetic user interface components, namely the round information display in the center, had to be implemented somehow. Initially, labels were placed in the middle of the screen with the use of control nodes, but the end result didn't look good, since the view had a 3D perspective to it while the labels were rendered as flat 2D text on top of it. Godot did not have support for 3D labels, so the round information had to be implemented using 3D sprites that used 2D text labels as their material. The material had to be dynamic, so that when the points of a player were to be updated, the sprite would display this change immediately. This was accomplished by using a viewport to create a texture based on its children text label nodes, which would be updated on every frame. The 3D sprites all used this texture as their material, but with different regions, effectively making it a dynamic sprite sheet. The end result ended up looking more fitting to the game world than with flat 2D text rendering, and gave a scent of realism to the game. This also showed the flexibility of Godot's viewport nodes. The process is detailed in Figure 31.

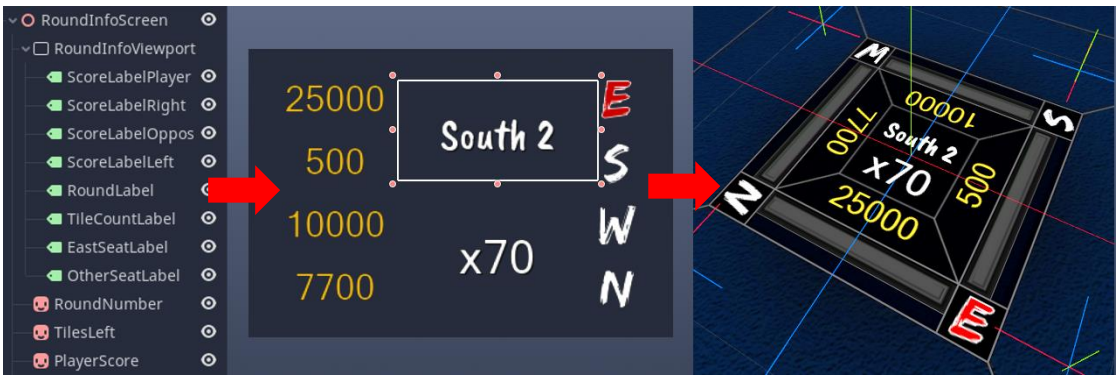


Figure 31. Using a viewport to project 2D labels into a texture, then rendering them into the game world using 3D sprites. Retrieved on 24 May 2021.

Finally, lighting was set up for the game world. There were three types of light nodes available in Godot: Directional lights, Omni lights, and Spot lights. Directional lights are the cheapest form of lighting in terms of performance, and emit light in one direction, comparable to the Sun when viewed from Earth, and were used partially to light up the player's own hand. Omni lights emit light spherically in all directions, and were placed above the center point of the mahjong table to give an even shadow to all the tiles. Spot lights emit light in the form of a cone, but they weren't used for the project. As can be seen for example in Figure 29, Godot's default environment lighting looks quite bland. Environment settings can be tweaked in order to change contrast, saturation and brightness of the scene, and certain effects like screen-space ambient occlusion, glow and fog can be enabled from there. In addition, environments provide ambient lighting over the entire scene, and can also have included sky attributes, which cast additional sky-like natural lighting. The contrast and saturation were slightly raised, and the background color was changed to black. For some reason, the other effects didn't work very well with the tile objects, so they were left disabled.

Since objects in Godot's game worlds can reflect light onto other objects, another major way to enhance lighting is by using GI Probes, or Global Illumination Probes. GI Probes provide pre-baked lighting effects on static objects, after which they emit real-time, dynamic indirect lighting based on existing light sources. If the "Interior" setting is checked in the GI Probe settings, it will also ignore any ambient lighting

from the scene's environment or the sky. In order for the mahjong tiles to cast shadows, the respective setting had to be enabled in its mesh settings first. A GI Probe was then applied over the mahjong table, which was done simply by scaling the probe box around it, and the Interior setting was enabled, after which the probe's lighting was baked. This caused the table borders and/or mat to reflect light on the backsides of the tiles, smoothing up some rough shadows, and looking generally better without ambient lighting. The results can be seen compared in Figure 32, where modified environment settings and omni lights are used respectively with and without additional GI Probe lighting.

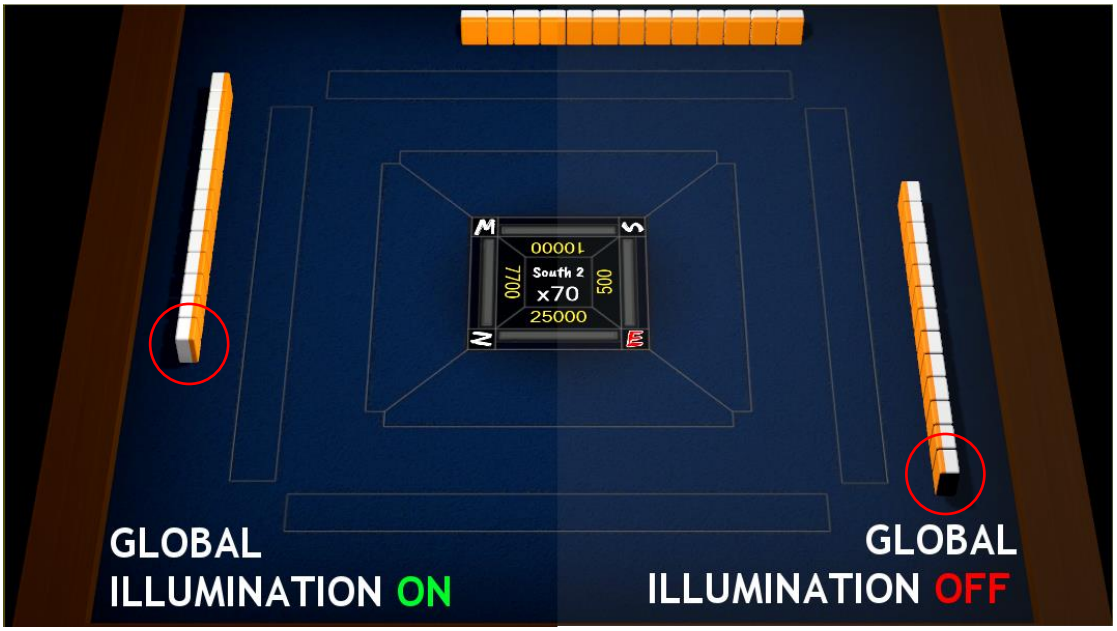


Figure 32. GI Probe results compared. Notice the smoothing of rough shadows circled in red. Retrieved on 24 May 2021.

5.7 Implementing multiplayer using a server-client architecture

At first, the game was run with a single player, while the rest of the player slots were filled with bots that would merely discard a random tile on their turn. Godot's networking features were then taken advantage of to add multiplayer functionality to the game. Due to the way that the game's code was designed, the architecture was there from the beginning, so all that had to be done was to separate the game into

two executables: the client, and the server. Clients would essentially contain the visible part of the game, while the entire game logic would be running on the server, which is a good way to prevent cheating if the game's multiplayer were to be later expanded upon.

Setting up the client and server in Godot was surprisingly easy. All the code required for setting them up can be seen respectively in Figure 33. As can be seen, there aren't many differences between the two. The IP "127.0.0.1" refers to localhost, since the server was run on the same host machine as the client in this case, but the server could be run from anywhere, in which case only the IP would have to be changed. The multiplayer connection uses the request-response protocol (RPC) by default, which is based on the client-server model. RPC supports both TCP and UDP networking protocols, of which the TCP implementation was used for the project. Upon receiving a connection, a random RPC ID is assigned to the client, which can be used to communicate back to them. It's noteworthy to mention that the protocol is currently not encrypted by default, so care should be taken before planning large-scale game projects using it, even though network encryption for high-level multiplayer is planned for Godot version 4.0 (Nations, 2021). Alternatively, Godot also offers WebSocket, SSL and HTTP protocol support for networking purposes (Godot Docs – High-level multiplayer).

Figure 33. Server and client multiplayer setup in Godot. Retrieved on 24 May 2021.

As Godot's high-level multiplayer is based on the node tree system, both the server and client classes had to be named the same (in this case "Server"), and initialized as singletons from the Project Settings' AutoLoad tab. As singletons, the Server node in both cases is initialized on top of the node tree, so they're on the same level and are able to communicate with each other, and are also globally accessible even if the active scene is changed. RPC requests had to be made from the same level, which would attempt to call a method on either side, such as "ReceiveClientConnection" or "ReceiveDataFromServer". In order for the method calls to work, they had to be marked as remote, which is done in C# by adding a "[Remote]" tag above the method. Methods have to be explicitly marked remote like this, as otherwise both the client and server would be able to call any existing method on either side, which could be used maliciously. A map of the implemented network architecture and its flow can be seen in Figure 34.

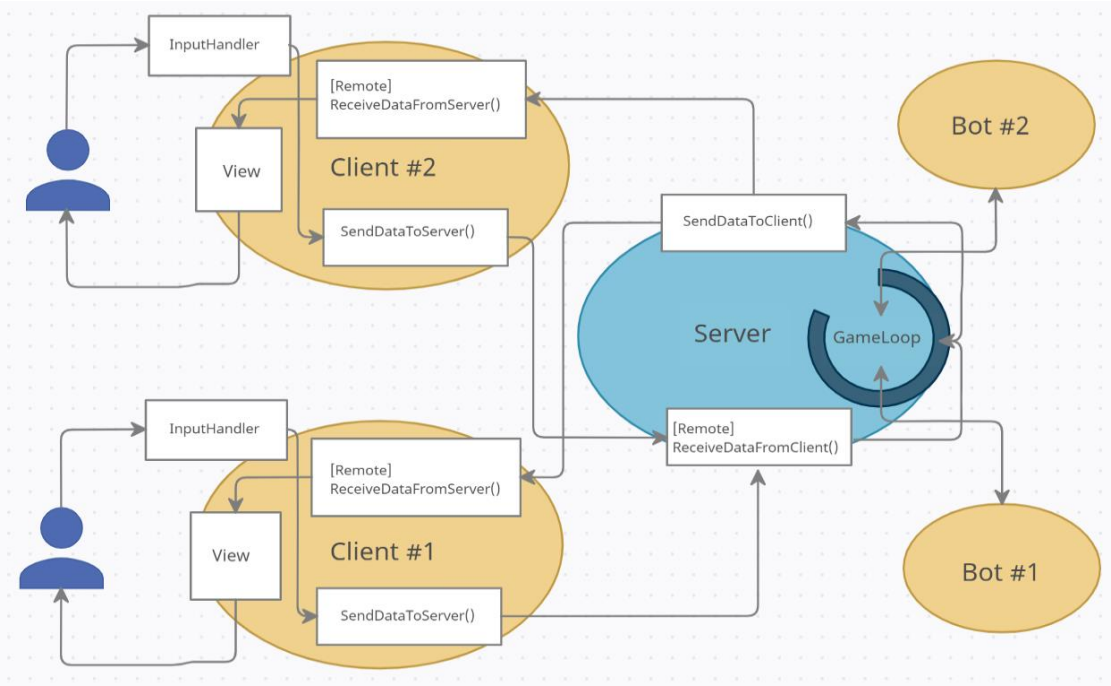


Figure 34. Server-client multiplayer architecture implemented using Godot networking features. Retrieved on 24 May 2021.

The network functionality was tested by connecting two clients to the server at the same time. The clients were assigned a client handler to keep track of their player instances on the game loop, which would also let disconnected players reconnect to the game. The other two players were replaced with bots. After making the clients play a round of mahjong against each other, the multiplayer experiment was marked as a success. An excerpt of the server log can be seen in Figure 35, where players Alice and Bob successfully connect to the server and are assigned a random client ID. The client handler IDs are different, because the client's own ID is a constant unique identifier that was coded in, while the received RPC connection ID is always variable and randomly generated from inside Godot, and is merely used by the server to keep track of its connections.

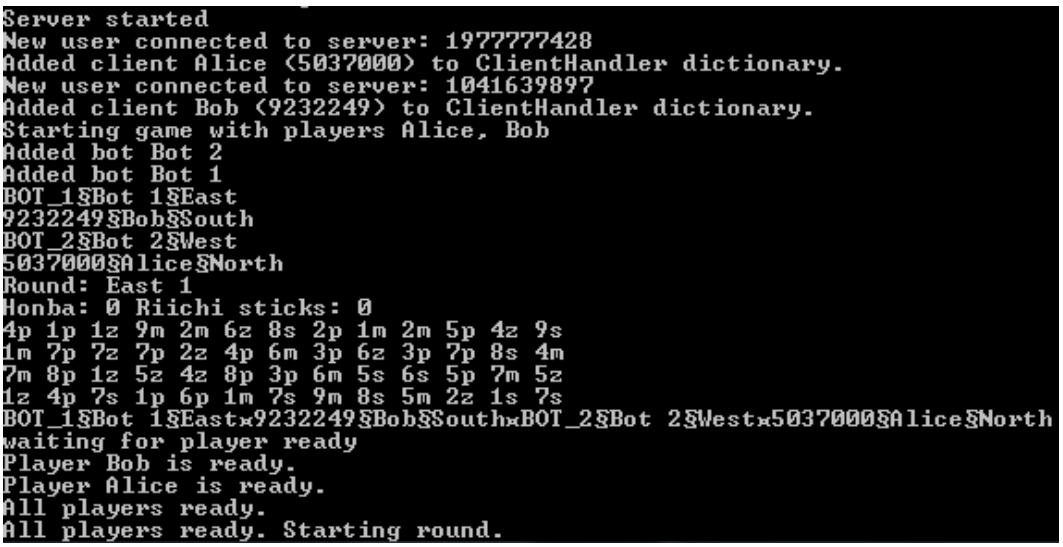


Figure 35. Excerpt of the implemented server's log, showing two successfully connected players. Retrieved on 24 May 2021.

5.8 Exporting the game

Before exporting the project, some of its project settings were tweaked, including window size and fullscreen mode. Some of the quality-related settings were also configured, such as the level of MSAA (multisample anti-aliasing) for smoothing the edges of the mahjong tile objects. Both the server and the client were then exported from the export menu. The engine offered exporting the project as is, as well as in PCK format, as seen in Figure 36. The exportable platforms were the same as advertised on Godot's website. The PCK format can also be used to pack specific scenes or other resource files in it. The purpose of exporting in PCK is to be able to provide end users with patch files and additional downloadable content, so that the entire project doesn't have to be re-downloaded to be up to date. It can also be used by users to create mod content for the game, as long as the developer provides an API for them to do so. (Godot Docs – Exporting packs, patches, and mods.)

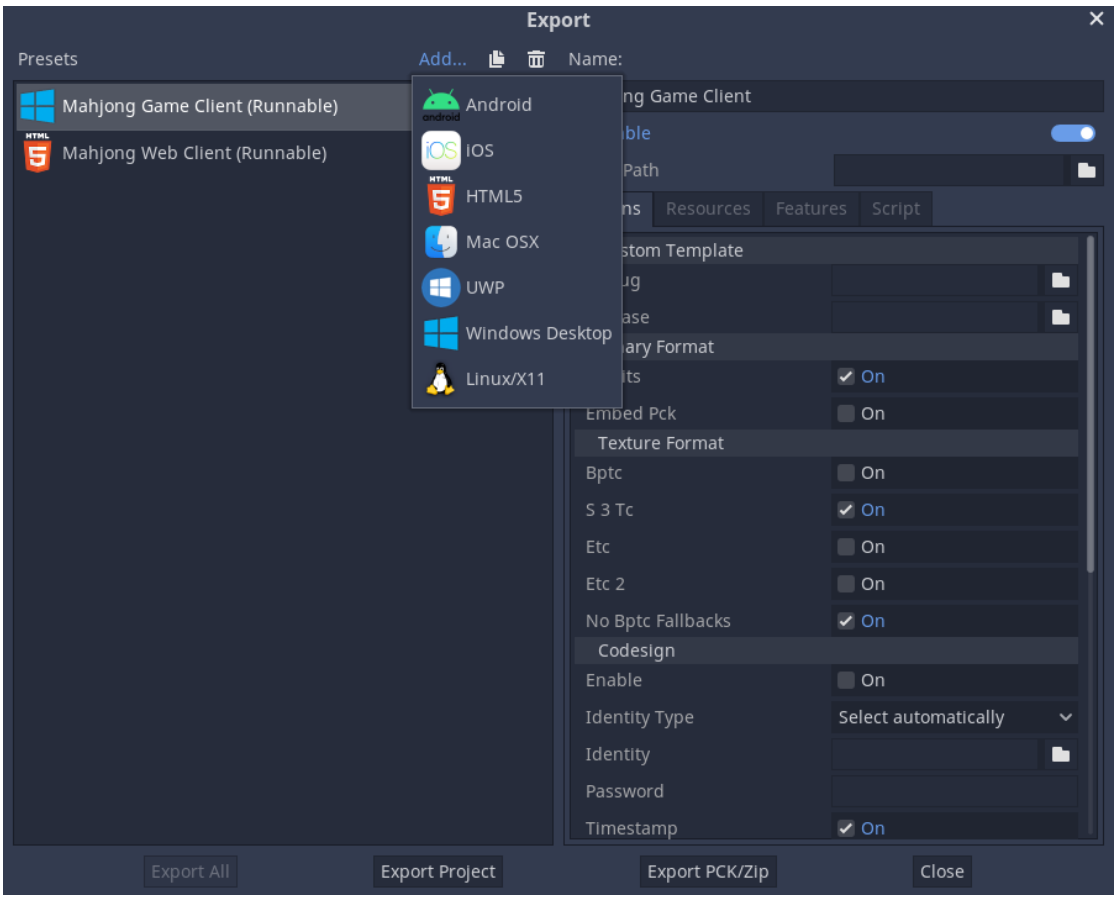


Figure 36. Project export options in Godot. Retrieved on 24 May 2021.

Before being able to export, an export template file specific to the target platform had to be downloaded (Figure 37) so that the engine would know how to pack the project. For the Windows target platform using C#, the file was around 560 megabytes in size. This is good to keep in mind in case the developer has to export their project in an environment with slow internet speeds, without having pre-downloaded the template file. Exporting onto Windows ended up with three files: the binary executable, a PCK file with the resources, and a DLL file for the Git API that was imported during project setup, as well as one folder containing C# Mono configuration files.

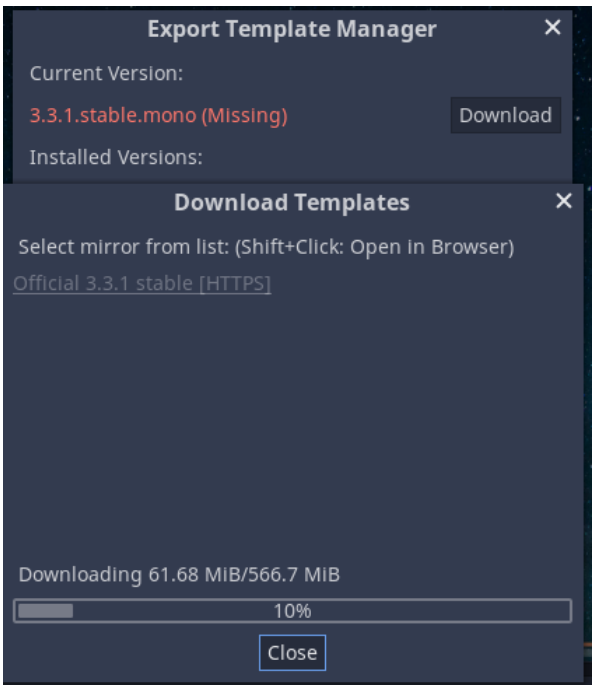


Figure 37. Downloading template files in Godot for exporting. Retrieved on 24 May 2021.

6 Results

As an end result, a working 3D mahjong game prototype with multiplayer functionality was produced using Godot Engine. The game was programmed using .NET Framework's C# language, with over 3,500 lines of code produced in total. The main Godot version used during the development process was 3.2.3, released on September 17, 2020. Later updates to Godot, versions 3.3 and 3.3.1, were applied upon release on April 21 and May 18, 2021, respectively. The final result can be seen in Figure 38, and represents Godot version 3.3.1. In general, Godot performed well during the development process. Without prior knowledge of the engine, it was generally intuitive to learn and to work with. Its documentation was extensive, and helpful during the development process. Aside from a few short slowdowns when instancing lots of objects at the same time, the engine performed smoothly during gameplay. Barely any crashes happened with the editor, either.

Figure 38. Final version of the mahjong game prototype. Retrieved on 24 May 2021.

C# development in Godot took time to get used to when coming from an OOP background. Since every Godot scene is a class in C#, it wasn't very intuitive that the class constructor wasn't enough to instantiate the scene, but rather the scene would also have to be added as a child of some node. It also wasn't possible to pass C# classes as signal parameters, if they didn't inherit from Godot's own Object or Node class. This lead to the impression that Godot's node system is somewhat rigid with C#, in the sense that everything has to be a node in order to comply with the engine and work properly. While it was certainly possible to use classes in code that didn't inherit from Godot, many errors were still caused by bugs like forgetting to add a constructed class as a child node, which tried to use Godot-specific features like signals or timers. In summary, it can be said that developers should be prepared to use nodes when developing with C#, and to play by Godot's rules if they want to have an easy time. This node-based workflow is most likely more intuitive in the official GDScript language, as the engine is built around it, while C# is not yet fully supported.

Aside from this, Godot's node and scene system from the editor's perspective was very modular, but also intuitive to work with, and fast to learn the basics of. Scene

instancing especially came into use when tile scenes were instanced into player hand scenes, and the hand scenes were then instanced into the game world. 3D game world creation was thus very doable in Godot. The vast amount of different nodes also provides many options for developing multiple types of games, which is important for game engines. Different kinds of control nodes were especially made use of during user interface design, and having parent and children nodes in this context made it easy to set up flexible spacing and positioning for UI elements. Godot is therefore also a good engine for designing game user interfaces. Tile discard animations were implemented comfortably using Tweens, which were used to interpolate translation and rotation properties. For the purpose of mahjong, all the nodes were sufficient, with the possible exception of 3D labels that were lacking, but using viewport textures was an efficient solution to this problem.

Viewport nodes sufficiently solve 3D world problems like 2D text rendering in 3D and provide split-screen features, like seen with the mahjong hand display. The encountered problems of displaying the player hand on top of the main screen, as well as displaying the center screen text from a 3D perspective, were both solved using viewport nodes. Their flexibility makes the engine more reliable in solving edge case problems like these. The engine's lighting options were sufficient for creating appropriate lighting for 3D games, including high-quality dynamic shadows. Shaders were also more than viable to develop in Godot. The live preview made coding them much more intuitive and faster than without any preview. VisualShaders and their non-programmer friendly workflow also make Godot an all-around engine for even the artists of the development team, but they cannot be relied on for delivering every kind of desired effect. As for the multiplayer, it was fast and simple to implement, easily showcased just by Figure 33, and works well as long as the node tree is set up correctly. Exporting the project was also straightforward, and the produced executable worked as expected.

Since Godot is an open-source game engine, even if a solution to a problem cannot be found, it's still possible for the developers to create one. However, for the purpose of 3D game development, the evaluation of which was the goal of the study, general 3D game development features are expected to be provided with the engine.

This is because not nearly every development team has the time to implement the functionalities necessary to make the game engine capable of 3D game production themselves. In other words, just by being open-source doesn't mean that the engine is a good platform for 3D development, even if it is a beneficial freedom to have. In Godot's case however, nothing excessively difficult had to be implemented in order to develop the mahjong test case project with it. This, along with the combined results of all the tested features, shows that Godot is indeed a viable game engine for 3D game development. Based on this, it's also a decent alternative to other engines like Unity or Unreal Engine, when it comes to developing 3D games the size of mahjong. However, its node-based workflow takes some time getting used to when developing with C#.

7 Discussion

The goal of the study was to determine the viability of using Godot Engine for 3D game development. Development of a 3D mahjong game was used as the test case for determining this. While the results show that Godot is indeed a viable engine for 3D development, this should be taken with a grain of salt. The 3D development was only tested for a project the size of mahjong, which gives no guarantee that Godot is also a valid choice for developing much larger 3D projects than mahjong. As claimed by Lovato, it could indeed be that it struggles with large game worlds, which cannot be determined from this study alone. Therefore, further studies need to be made in order to see how capable Godot is in handling larger 3D projects. Despite this, over 3,500 lines of code were still written for the project over the course of four months, so the scale of it does give a general idea of what it's really like to work with Godot on a 3D game, as opposed to following an online step-by-step tutorial, where real problems during development don't get the chance to occur.

Another, perhaps minor reliability issue of the study comes with the tech industry in general, namely with its advancement and the static nature of books and studies. Many things from this study could be invalidated just with the release of a new ver-

sion of Godot. An example is the FBX importer, which was rewritten between versions 3.2.3 and 3.3. If the importer had been assessed based on version 3.2.3, the result would've become obsolete within 7 months, which is what happened with Brandt's claims regarding the same importer. This simply means that claims made in this study might not be applicable to newer versions of the engine, especially with Godot version 4.0 coming out in the near future, and must be isolated to the versions in which the study was carried out with. The main result however, is unlikely to change, and rather 3D development in Godot should become even more efficient with future updates.

With regard to documentation, one of the most useful things about it (in my case) was that code snippets were available in both GDScript and C#, which made development faster, as there was no need to translate code. At no point did it feel that an answer could not be found either from the documentation, or from the pool of community tutorial content, although some language localizations of the documentation were very lacking in content. Despite Hewer claiming that there isn't that much tutorial content available for Godot, there was even more content available than expected, in form of both YouTube videos and text-based guides. Based on the few interactions that I had with the community in Godot's official Discord channel, the community came across as positive, team-spirited and eager to help beginners. It can be assumed that one contributing factor to the community's spirit is thanks to the engine's open-source nature, making the entire development of the engine a community effort.

There are certainly features of the engine that could've used more in-depth exploration. Some main features, like the audio and physics engine, weren't used for the project's prototype. Audio had least priority, and had to be cut due to time constraints. Physics on the other hand, weren't particularly needed for the game at this stage, as objects were simply translated to their target positions. Not every feature fit the scope of the project, which is why they serve as material for future studies. Meanwhile, studies related to Godot's use of viewports and its multiplayer functionality, as well as its C# workflow haven't seemingly been made before until now. Viewports are a feature that were run into just to solve specific problems, after all.

Multiplayer, however, can definitely be expanded upon. The mahjong project uses a server-client architecture, but is capped to four players and turn-based gameplay due to the game's nature. Hence, it wasn't possible to go very in-depth on Godot's multiplayer features, but aspects like authorization and server data management, protocol testing, and real-time multiplayer with dozens of players are interesting topics for future studies to cover. I believe this study's results can be used as sources of motivation for more in-depth studies regarding multiplayer in Godot, ideally featuring larger 3D worlds. It should also give a general idea of what to expect when choosing to develop games in Godot using the .NET Framework, which can be important to know for teams migrating to it from Unity.

Godot's future prospects are looking bright. With version 4.0 being released in the near future—providing support for the Vulkan API—Godot might become a great rival to even the most popular engines like Unity. Its open-source nature and lenient MIT license are great incentives of growth for its community. Once more bigger budget games are published under it, the engine should be able to properly take off in popularity. Due to how intuitive its node and scene system is, I think it could even be used for the purpose of teaching children the basics of game development. Concerning the mahjong game project, the source code will be kept private for now, as I have plans on making a valid product out of it over time, though whether it's going to be monetized is uncertain. In any case, the current version has some cut content and bugs left in, which make it a good hobby project to work on in my pastime.

As a final note, I would like to acknowledge my supervisors Jani Immonen and Esa Salmikangas for the feedback given on this thesis.

I would also like to thank Leevi Kukkonen, the artist that helped with the project, for the creation of the tile mat texture seen in many of the figures.

Lastly, I would like to thank my father Jorma Mäkelä, and friends Niels and Michele for the feedback and support given during the writing of this thesis, which had a positive influence on its quality.

References

Adams, E. (2010). *Fundamentals of Game Design*. 2nd ed. New Riders.

Baker, M. (2016, November 2). How Do Game Engines Work? *Interesting Engineering*. Accessed on 21 May 2021. Retrieved from https://interestingengineering.com/how-game-engines-work

Batchelor, J. (2018, March 20). Crytek adopts royalties model as CryEngine 5.5 arrives. *GamesIndustry.biz*. Accessed on 22 May 2021. Retrieved from https://www.gamesindustry.biz/articles/2018-03-20-crytek-adopts-royalties-model-as-cryengine-5-5-arrives

Brandt, J. (2020, December 4). Creating Games using the Godot Game Engine. *Making Games*. Accessed on 22 May 2021. Retrieved from https://www.making-games.biz/programming/creating-games-using-the-godot-game-engine,2351664.html

Dealessandri, M. (2020, April 15). What is the best game engine: is Godot right for you? *GamesIndustry.biz*. Accessed on 25 February 2021. Retrieved from https://www.gamesindustry.biz/articles/2020-04-14-what-is-the-best-game-engine-is-godot-right-for-you

European Mahjong Association. (2016). *Riichi – Rules for Japanese Mahjong*. European Mahjong Association website. Accessed on 24 February 2021. Retrieved from http://mahjong-europe.org/portal/images/docs/Riichi-rules-2016-EN.pdf

FluffyStuff. (2016). *Vector graphics of riichi mahjong tiles* [list of PNG image files]. Accessed on 23 May 2021. Retrieved from https://github.com/FluffyStuff/riichi-mahjong-tiles/tree/master/Export/Regular

Gao, S., Okuya, F., Kawahara, Y., & Tsuruoka, Y. (2019, June 7). *Building a Computer Mahjong Player via Deep Convolutional Neural Networks.* Cornell University Library. Accessed on 24 January 2021. Retrieved from https://arxiv.org/abs/1906.02146

GitHub. *Godot Engine – Multi-platform 2D and 3D game engine*. Godot Engine GitHub repository. Accessed on 21 May 2021. Retrieved from https://github.com/godotengine/godot

Godot Assets Marketplace. *EULA and Distribution Agreement*. Godot Marketplace website. Accessed on 16 March 2021. Retrieved from https://godotmarketplace.com/eula-and-distribution-agreement/

Godot Docs – About the Asset Library. Godot Engine documentation. Accessed on 16 March 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/assetlib/what_is_assetlib.html

Godot Docs – Console support in Godot. Godot Engine documentation. Accessed on 25 February 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/platform/consoles.html

Godot Docs – Differences between GLES2 and GLES3. Godot Engine documentation. Accessed on 23 May 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/misc/gles2_gles3_differences.html

Godot Docs – Exporting packs, patches, and mods. Godot Engine documentation. Accessed on 24 May 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/workflow/export/exporting_pcks.html

Godot Docs – GDScript: An introduction to dynamic languages. Godot Engine documentation. Accessed on 11 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/scripting/gdscript/gdscript_advanced.html

Godot Docs – Getting started with Visual Scripting. Godot Engine documentation. Accessed on 4 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/scripting/visual_script/getting_started.html

Godot Docs – High-level multiplayer. Godot Engine documentation. Accessed on 24 May 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html

Godot Docs – Importing scenes. Godot Engine documentation. Accessed on 30 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/workflow/assets/importing_scenes.html

Godot Docs – Instancing. Godot Engine documentation. Accessed on 4 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/instancing.html

Godot Docs – Resources. Godot Engine documentation. Accessed on 24 May 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/resources.html

Godot Docs – Scenes and nodes. Godot Engine documentation. Accessed on 4 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/scenes_and_nodes.html

Godot Docs – Scripting. Godot Engine documentation. Accessed on 11 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting.html

Godot Docs – Shaders. Godot Engine documentation. Accessed on 23 May 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/shaders.html

Godot Docs – Signals. Godot Engine documentation. Accessed on 24 May 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

Godot Docs – Unit testing. Godot Engine documentation. Accessed on 11 March 2021. Retrieved from https://docs.godotengine.org/en/latest/development/cpp/unit_testing.html

Godot Docs – Version Control Systems. Godot Engine documentation. Accessed on 26 March 2021. Retrieved from https://docs.godotengine.org/en/stable/getting_started/workflow/project_setup/version_control_systems.html

Godot Docs – VisualShaders. Godot Engine documentation. Accessed on 23 May 2021. Retrieved from https://docs.godotengine.org/en/stable/tutorials/shading/visual_shaders.html

Godot Engine – Community. Godot Engine website. Accessed on 17 March 2021. Retrieved from https://godotengine.org/community

Godot Engine – Features. Godot Engine website. Accessed on 25 February 2021. Retrieved from https://godotengine.org/features

Hasu, J. (2018). *Fundamentals of Shaders with Modern Game Engines* [Master's thesis, Lappeenranta University of Technology]. LUTPub. http://urn.fi/URN:NBN:fi-fe2018110547290

Herzog, T. (2017, April 5). *GDNative is here!*. Godot Engine website. Accessed on 11 March 2021. Retrieved from https://godotengine.org/article/dlscript-here

Lavieri, E. (2018). *Getting Started with Unity 2018 - Third Edition: A Beginner's Guide to 2D and 3D Games with Unity*. 3rd ed. Packt Publishing.

Linietsky, J. (2017, August 3). *Why we should all support glTF 2.0 as THE standard asset exchange format for game engines*. Godot Engine website. Accessed on 30 March 2021. Retrieved from https://godotengine.org/article/we-should-all-use-gltf-20-export-3d-assets-game-engines

Linietsky, J. (2019, March 26). *Status of Godot at GDC 2019*. Godot Engine website. Accessed on 24 February 2021. Retrieved from https://godotengine.org/article/status-godot-gdc-2019

Linietsky, J. (2020, July 8). *Donation changes*. Godot Engine website. Accessed on 24 February 2021. Retrieved from https://godotengine.org/article/donation-changes

Linietsky, J. [@reduzio] (2021, May 20). *Just to clarify because this seems like a large source of misunderstandings. We _are_ working on a revamp of the* [Tweet]. Twitter. https://twitter.com/reduzio/status/1395380550983196673

Miettinen, T. (2019). *2D-pelin kehittäminen Godot Enginellä* [2D Game Development with Godot Engine]. [Bachelor's thesis, KAMK University of Applied Sciences]. Theseus. http://urn.fi/URN:NBN:fi:amk-2019052010596

Nations, W. (2021, February 20). Why You Should Pay Attention To Godot Engine In 2021. *Christian Game Developers Conference*. Retrieved from https://christiangamedevelopersconference.com/why-you-should-pay-attention-to-godot-engine-in-2021/

Schardon, L. (2021, January 2). Best Game Engines of 2021. *GameDev Academy*. Accessed on 22 May 2021. Retrieved from https://gamedevacademy.org/best-game-engines/

Spencer, G. (2019, August 29). More than a game: Mastering Mahjong with AI and machine learning. *Microsoft Stories Asia.* Accessed on 24 January 2021. Retrieved from https://news.microsoft.com/apac/features/mastering-mahjong-with-ai-and-machine-learning/

Stephenson, S. (2019, August 13). TIGA Survey Reveals that Unity 3D Engine Dominates the UK Third Party Engine Market. *TIGA*. Accessed on 22 May 2021. Retrieved from https://tiga.org/news/tiga-survey-reveals-that-unity-3d-engine-dominates-the-uk-third-party-engine-market

Tenhou.net. Japanese mahjong wiki arcturus.su. Accessed on 16 January 2021. Retrieved from http://arcturus.su/wiki/Tenhou.net

Verschelde, R. (2021, April 21). *Godot 3.3 has arrived, with a focus on optimization and reliability*. Godot Engine website. Accessed on 22 May 2021. Retrieved from https://godotengine.org/article/godot-3-3-has-arrived#fbx-importer

Zarrad, A. (2018). Game Engine Solutions. In *Simulation and Gaming*. InTech. Accessed on 22 May 2021. https://doi.org/10.5772/intechopen.71429