UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Hannogert Otti

Box Constructor – Greyboxing Tool for Godot Game Engine

Bachelor's Thesis (9 ECTS)

Supervisor:
Jaanus Jaggo, MSc

Tartu 2025

Box Constructor – Greyboxing Tool for Godot Game Engine

Abstract:

Greyboxing is an important part of the prototyping phase, allowing developers to test and experiment with their ideas before investing time into creating a detailed level. This thesis creates an intuitive greyboxing solution for the Godot game engine, addressing the need for a simple and user-friendly tool. This tool allows users to create simple levels straight in the editor without relying on external tools. Box Constructor streamlines the process by enabling quick and efficient methods for adding and removing geometry. The testing results concluded with a usability score of over 80%. The thesis concludes with recommendations for future enhancements to improve usability and add more functionality.

Keywords: level design, greyboxing, Godot engine, plugin

CERCS: P170 Computer science, numerical analysis, systems, control

Box Constructor – Hallkastimise tööriist Godot mängumootorile

Lühikokkuvõte:

Hallkastimine (greyboxing) on oluline etapp prototüüpimise faasis mis võimaldab arendajatel testida ja katsetada oma ideid enne detailse taseme loomist. Käesoleva lõputöö raames luuakse intuitiivne hallkastimise tööriist Godot mängumootirile, pakkudes lihtsat ja kasutajasõbralikku lahendust. See tööriist võimaldab kasutajatel luua tasemeid otse Godoti mängumootoris ilma väliste tööriistateta. Tööriist pakub tõhusaid meetodeid geomeetria lisamiseks ja eemaldamiseks. Testimistulemused näitasid, et kasutatavuse tulemus on üle 80%. Lõpus tuuakse soovitusi tulevasteks täiustusteks, et parandada kasutajamugavust ja lisada funktsionaalsust.

Võtmesõnad: tasemedisain, hallkastimine, Godot mängumootor, pistikprogramm

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1. Introduction 4
2. Level Design 6
3. Prototyping 9
3.1 Paper Prototypes 9
3.2 Digital Prototypes 10
3.3 Greyboxing 10
4. Overview of Existing Applications 12
4.1 Cyclops Level Builder 12
4.2 Grid Modeler 16
4.3 Cube Grid 18
5. User Experience 21
5.1 Design of the Toolbar 22
6. Implementation 23
6.1 Setup and Architecture 23
6.2 Grid and Snapping 24
6.3 Geometry Creation 25
6.3.1 Base Rectangle Drawing 25
6.3.2 Extruding 26
6.4 Corner Movement 29
6.5 Mesh Merging and Reconstruction 30
6.5.1 Merging 30
6.5.2 Reconstruction 31
7. Testing 32
7.1 UMUX 32
7.2 Participants 33
7.3 Usability Evaluation 34
7.4 Overall Experience and Preferences 36
7.5 Performance Testing 37
8. Conclusion 39
References 40
Appendix 41
I — Glossary 41
II — Accompanying Files 42
III — License 43

1. Introduction

Game design is an iterative process involving a lot of ideation, prototyping and testing. A crucial aspect of this process is level design, as levels are the primary way for players to interact with the game. To efficiently plan and refine level layouts, developers use rapid prototyping methods. One of the most used approaches is greyboxing, where draft-level layouts are built out of simple shapes such as cubes.

This thesis introduces Box Constructor, a user-friendly greyboxing solution for the Godot game engine. Figure 1 illustrates a simple greybox level created using Box Constructor. This tool enables users to create levels directly within the Godot editor. Box Constructor streamlines the geometry creation by allowing users to select an area of a surface and extrude it outward or inward to add or remove geometry. The main focus when developing the tool was on user experience and intuitiveness, ensuring a smooth and efficient workflow.

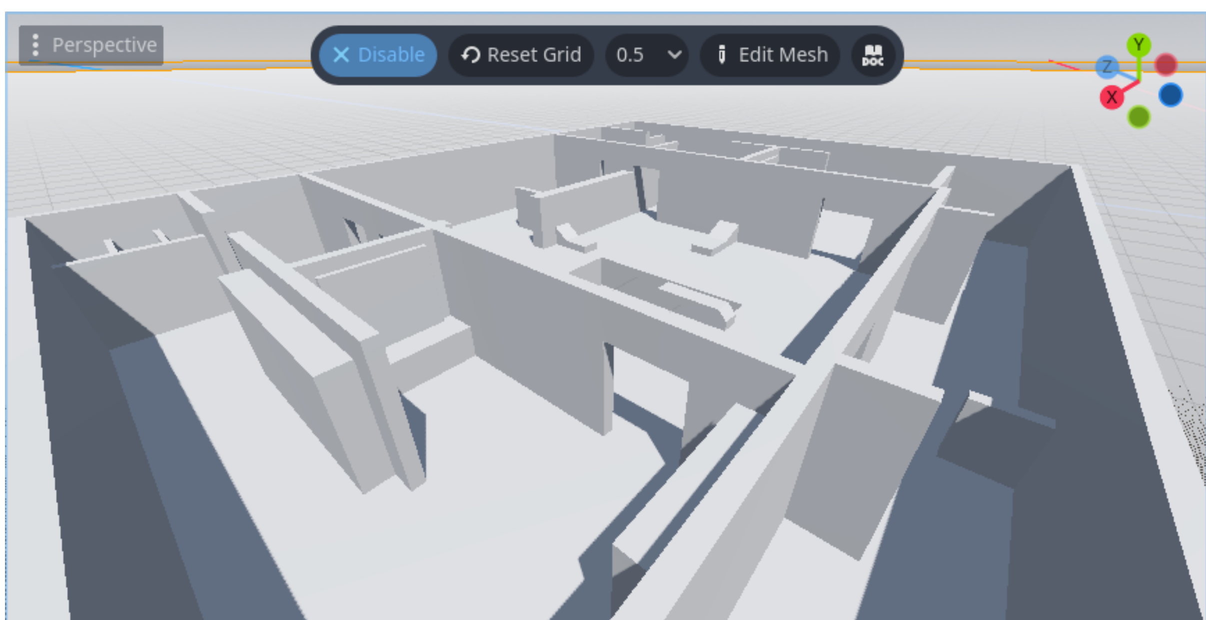

Figure 1. Example of a greybox level created in Box Constructor, inspired by the fy_poolday map from *Counter-Strike 1.6*[1].

[1] https://gamebanana.com/mods/86562

The main functional requirements of Box Constructor are:

1. Users can draw and extrude base shapes to create different-sized objects.
2. Users can move the edges of the created objects to create ramps and other slanted surfaces.
3. The grid system ensures that blocks are snapped within the grid.
4. Boolean operations are decided automatically based on the extrusion direction.
5. Users can merge all *Constructive Solid Geometry* (CSG) nodes into a single MeshInstance3D for better performance during gameplay testing.
6. Users can revert the merging to edit their objects further.

Performance testing was conducted to assess the usability and user-friendliness of the tool. These tests helped to identify how Box Constructor compares to other existing tools. After testing, the tool was uploaded to the Godot Asset Library[2], making it available to all Godot users.

The thesis is structured as follows:

1. Chapter 2 covers the fundamentals of game design.
2. Chapter 3 explores prototyping methods used in the industry.
3. Chapter 4 analyses existing greyboxing solutions, evaluating their strengths and weaknesses.
4. Chapter 5 discusses the user interface of Box Constructor.
5. Chapter 6 details the implementation and functionality of Box Constructor.
6. Chapter 7 presents testing methodology and results.
7. Chapter 8 concludes the thesis.

To enhance the clarity, coherence and readability of this thesis, Grammarly[3] was used to improve sentence structure and expression.

[2] https://godotengine.org/asset-library/asset/3944
[3] https://app.grammarly.com/

2. Level Design

Before discussing level design, it is important to understand what a level is. According to the definition from The Level Design Book, a level is a space where a game takes place, which defines the boundaries for player movement and interaction [1]. When most people think of a level, they might think of a stage in Super Mario, where the player progresses through a linear path (see Figure 2). However, levels can have many different forms. A game of *Solitaire* can also be considered a level, or the big open world of Grand Theft Auto can be seen as one large level.

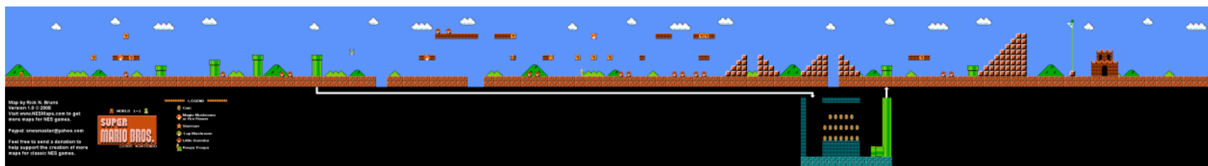

Figure 2. Level 1-1 from Super Mario Bros[4].

Level design, in turn, is the process of creating these playable spaces, providing the players with something to discover and an opportunity to learn the game's core mechanics. The purpose of level design should be to create worlds that keep the players engaged and immersed. According to *Practical Game Design* [2], game worlds can range from realistic to fantastical, only achieving believability through constant design iterations. The authors also highlight the importance of balancing realism and playability. Realistic elements can help players understand the game's core mechanics, while excessive realism can negatively impact the game's enjoyment. Therefore, excessive realism can complicate gameplay, forcing players to keep track of many little details. For example, designing a game world with realistic architecture, such as an office building or a shopping mall with multiple floors, numerous small rooms, and long winding hallways, makes the place feel realistic. However, it can also make the navigation of such a place frustrating for the players. They might struggle to find objectives or get lost due to complex layouts, like how players often find themselves lost in the maze-like world of *Dark Souls*, where in Blighttown (see Figure 3), the navigation is intentionally challenging to enhance immersion and to encourage exploration.

[4] https://www.nesmaps.com/maps/SuperMarioBrothers/SuperMarioBrosWorld1-1Map.html

Figure 3. Blighttown from *Dark Souls*[5].

Additionally, excessive realism can significantly increase development costs, as designers have to implement intricate details that may not contribute to player enjoyment. Therefore, the primary goal of level design is to craft immersive and believable worlds that enhance gameplay without overly restricting or overwhelming the player [2].

The game's world and level structure are crucial in shaping player engagement. As Totten states in his book *Architectural Approach to Level Design*, worlds created randomly or without planning often result in shallow experiences driven by ideas like "Wouldn't it be cool if..." [3]. To avoid this, designers should follow some guidelines for creating levels. As outlined by Kramarzewski and De Nucci [2], the process typically involves:

1. defining a starting point;
2. creating a sketch;
3. greyboxing;
4. adding artistic elements;
5. final polishing.

According to the authors, the first step is to define the basic principles that guide the course of the game and introduce players to the core mechanics. This step mainly focuses on the type

[5] https://darksouls.fandom.com/wiki/Blighttown

and objective of the game. Defining the starting point is important to understand the level's visual layout and achieve the intended gaming experience. The next step is to create a plan or a sketch, which may include anything from a rough level draft to dialogue between characters. The third step in the process is greyboxing, which aims to create a playable prototype level and define its basic structure and objects to test its playability. The second-to-last step is applying artistic elements to the level, such as adding lighting, textures and other details like foliage, to make the level feel more lifelike. After that, the process is complete, and the designers add final polishing touches to the level.

To conclude this chapter, the best way to go about level design is to follow a structured process, such as the one discussed above. By following principles set by Kramarzewski and De Nucci, designers can create game worlds that feel immersive and cohesive, which in turn contributes to the overall player enjoyment and the success of the game.

3. Prototyping

In game development, prototypes are created to experiment and test different ideas. According to Kramarzewski and De Nucci, the primary purpose of a prototype is to answer questions such as: "Does it work? " or "Is it engaging?" [2]. Prototypes help to identify design problems in the early stages of development. This process reduces risks and prevents having to start over. Hence, prototypes must be created in a way that allows for refinement and improvement, enabling quick adjustments and fixes throughout the process. When creating a prototype, the primary focus needs to be on the essential elements, such as gameplay mechanics. Different prototyping methods are used in game development, but the most common ones are paper and digital prototypes. The following subchapters discuss different prototyping methods used in the industry. Chapter 3.1 discusses paper prototyping while Chapter 3.2 discusses digital prototyping. Finally, Chapter 3.3 gives an overview of greyboxing.

3.1 Paper Prototypes

As the name suggests, paper prototypes involve sketching the game's ideas on paper or using cards. For example, the sketch can be a basis for testing inputs in a real-time strategy game or a layout of a level in a shooter game (see Figure 4). While digital tools are often used to simplify and speed up the process, paper prototypes still have many use cases in game development.

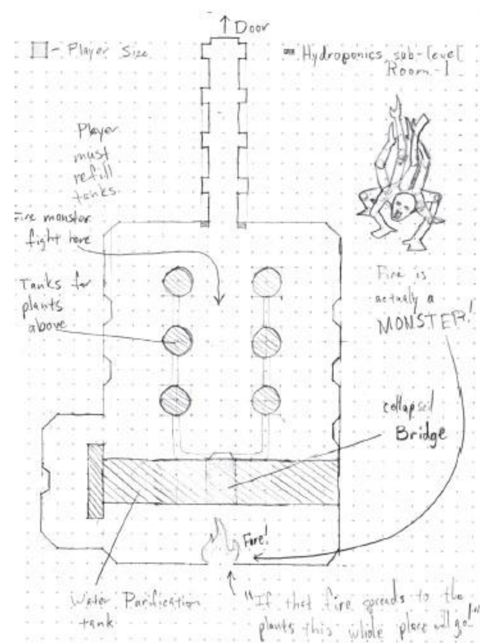

Figure 4. Example of paper prototype of level layout [3].

Jeremy Gibson Bond [6] highlights several benefits of paper prototyping compared to digital tools. One key advantage is its low technical barrier for entry, requiring little to no technical expertise, making it accessible to anyone. Additionally, designers do not need to pay much attention to details, allowing them to focus on core game mechanics mainly. Furthermore, paper prototypes can also be useful for designing Graphical User Interfaces. Printing out and drawing different mockups is quick and easy. However, there are still some drawbacks to using paper prototypes. For example, tracking complex calculations, such as damage numbers in a game, can be difficult using paper prototypes. Another significant

drawback is the inability to evaluate the control schemes for gamepads, keyboards and touchscreen inputs.

3.2 Digital Prototypes

Creating digital prototypes in game development involves using various digital tools to make the prototyping more efficient. Kramarzewski and De Nucci highlight that *CAD software*, such as Autodesk AutoCAD[6] and DraftSight[7], can create more precise and cleaner drawings than hand-drawn sketches [2]. These tools allow for creating more complex level layouts and help visualise the placement and dimensions of elements in the level, with the added benefit of being able to edit them quickly. The precision offered by CAD software is convenient when designing larger and more intricate levels, where the placement and proportionality of each element is important. Furthermore, Kramarzewski and De Nucci [2] state that art programs like Adobe Photoshop[8] or GIMP [9]can be used to add colour and lighting to designs, which helps convey the level's atmosphere and visual style.

Some game engines offer built-in tools for level design. For example, Unreal Engine's Cube Grid[10]. This tool provides a convenient way to construct basic level layouts directly within the game engine. This allows designers to see the final result and test it immediately. However, these tools often have limitations in modeling functionality and are primarily efficient for making basic structures.

In contrast, external modeling programs such as Blender[11] and Autodesk Maya[12] offer a broader range of modeling functions, enabling the creation of highly detailed assets with textures and complex shapes. The trade-off is that models created in these external tools require additional steps for importing and optimising within game engines.

3.3 Greyboxing

Greyboxing, or blockout, is a stage in the level design process where designers create a simple, playable version of the game level. In this stage, simple geometric shapes, known as primitives (e.g., cubes and spheres), outline the level's layout and core elements related to the game's

---

[6] https://www.autodesk.com/
[7] https://www.draftsight.com/
[8] https://www.adobe.com/products/photoshop.html
[9] https://www.gimp.org/
[10] https://dev.epicgames.com/documentation/en-us/unreal-engine/cubegrid-tool-in-unreal-engine
[11] https://www.blender.org/
[12] https://www.autodesk.com/products/maya/overview

mechanics. These shapes are typically untextured and grey, which gives the method its name [1]. The purpose of greyboxing is to create a practical version for testing playability, rather than a detailed environment (see Figure 5).

Figure 5. Comparison of greybox and the final product of Modern Warfare's "Docks" map[13]

Additionally, greyboxing is cost-effective. Designing detailed environments can be time-consuming and expensive. Discovering flaws in later stages of development often leads to costly rework. By enabling early playtesting, greyboxing removes this risk. Testing the greybox version helps identify structural weaknesses, such as pacing issues, allowing designers to address them early before investing in detailed design. In the next chapter, we will explore various applications for greyboxing and compare their workflows, strengths, and limitations.

[13] https://x.com/JrBakerChee/status/1182384066881916928

4. Overview of Existing Applications

Different software solutions have been created for game development to help developers and designers create functional game levels efficiently. This section provides an overview of such tools and analyses their features, advantages, and disadvantages to gain insight that helps to enhance Box Constructor.

The next subchapters provide an overview of the three tools:

1. Chapter 4.1 reviews Cyclops Level Builder, a *plugin* created for the Godot game engine.
2. Chapter 4.2 reviews Grid Modeler, an add-on for Blender created for hard surface modeling.
3. Chapter 4.3 examines Cube Grid, a native tool to the Unreal Engine.

4.1 Cyclops Level Builder

Cyclops Level Builder is an add-on for the Godot game engine designed to simplify the level creation process. The tool allows users to quickly create blocks by dragging a shape onto the grid and extruding it into a 3D object. It also includes a material editor for assigning textures and materials to blocks. Each block also features built-in collision for immediate level testing. With these additional features, Cyclops Level Builder is not only limited to prototyping, but can also be used to create a finalised game level.

To activate Cyclops Level Builder, users first need to select it from the bottom menu in the Godot editor (see Figure 6). After clicking "Create Block" in the opened menu, a new block is added to the scene. The user can start editing by selecting the block from the Scene Tree panel. Once selected, a toolbar appears, providing different options. Users can create new primitives

like cubes, prisms, cylinders or custom shapes like stairs. The taskbar also includes buttons for manipulating the shape of the created object (see Figure 7).

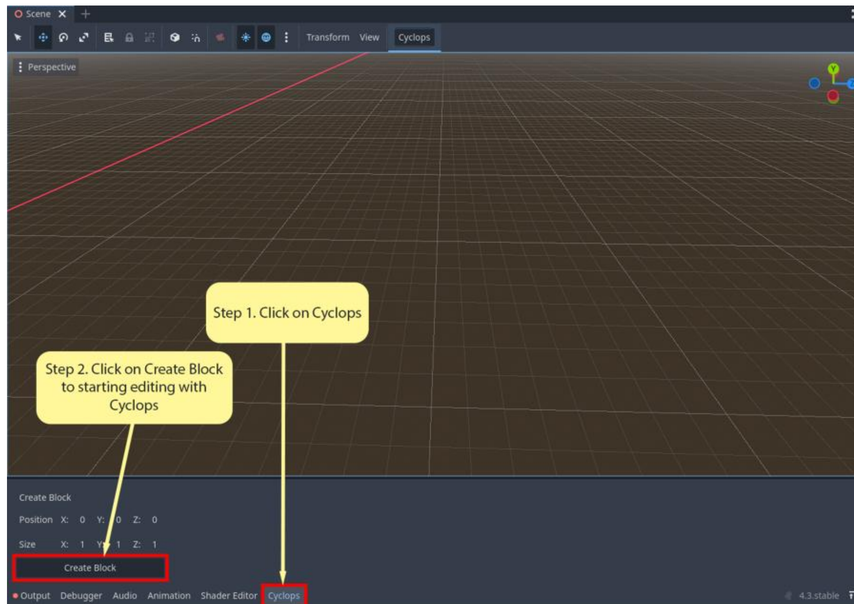

Figure 6. Adding the first block to Cyclops Level Builder.

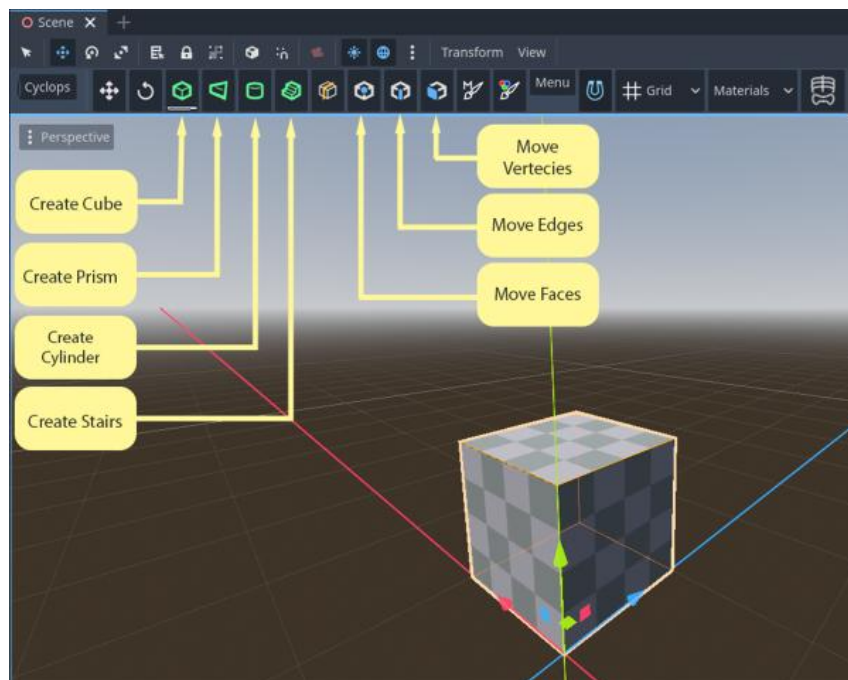

Figure 7. User interface of Grid Modeler.

To manipulate faces, vertices or edges of an existing 3D object, the user can select the corresponding manipulation method from the toolbar. After that, the user can click on the shown markers to create a *gizmo* at that location. Users can then drag the gizmo to manipulate the

chosen face, vertex or edge (see Figure 8). This feature gives users more control over their shapes and allows them to create more complex and interesting shapes.

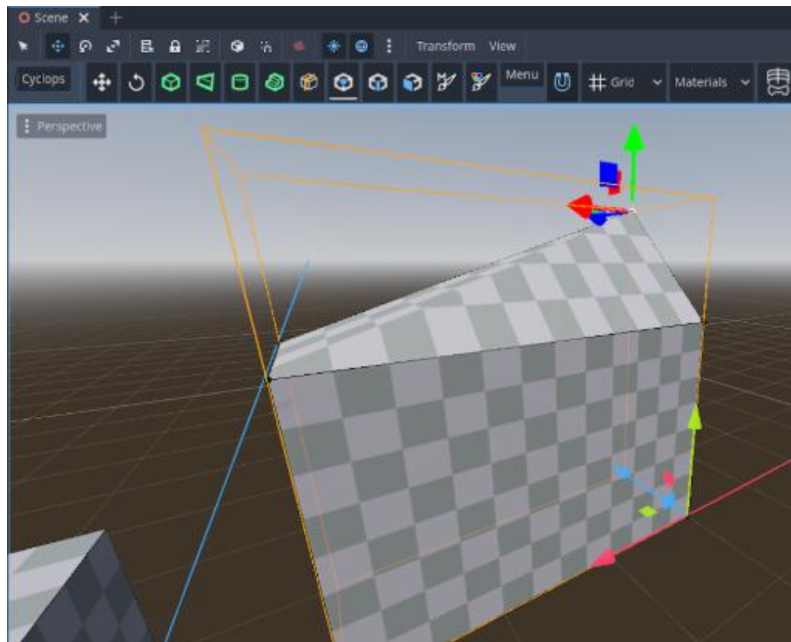

Figure 8. Vertex movement tool in Cyclops Level Builder.

To add new geometry, users can select the desired primitive shape from the toolbar and drag out its size on the grid; this defines the base shape of the 3D object. Upon releasing the left mouse, the shape begins to extrude in the direction of the mouse (see Figure 9). Clicking again finalises the extrusion, transforming the drawn 2D shape into a 3D shape.

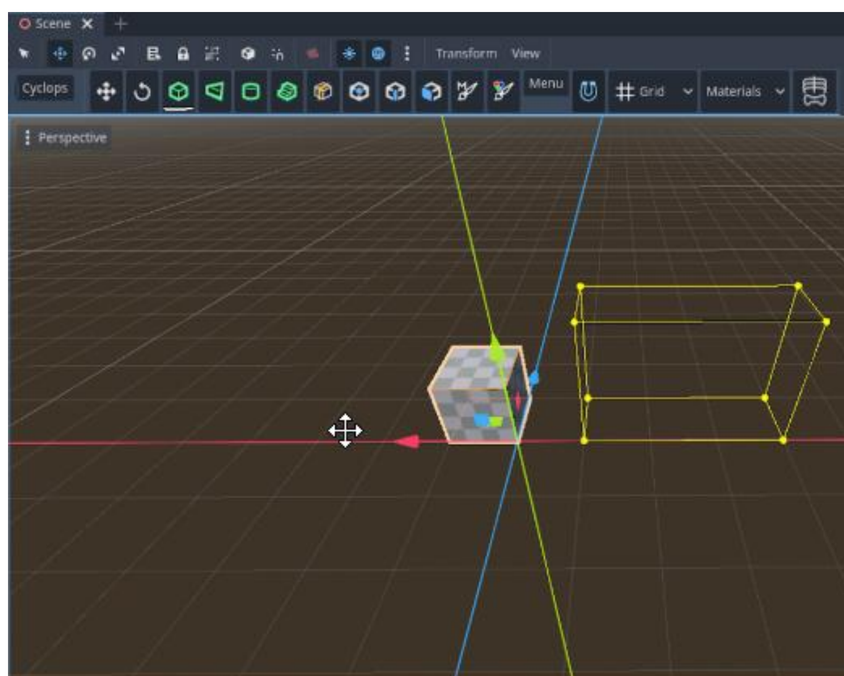

Figure 9. Extrusion preview in Cyclops Level Builder.

The add-on also supports Boolean operations, allowing users to subtract one block from another. To perform Boolean operations, users select the two blocks and use the "Subtraction" option from the context menu (see Figure 10).

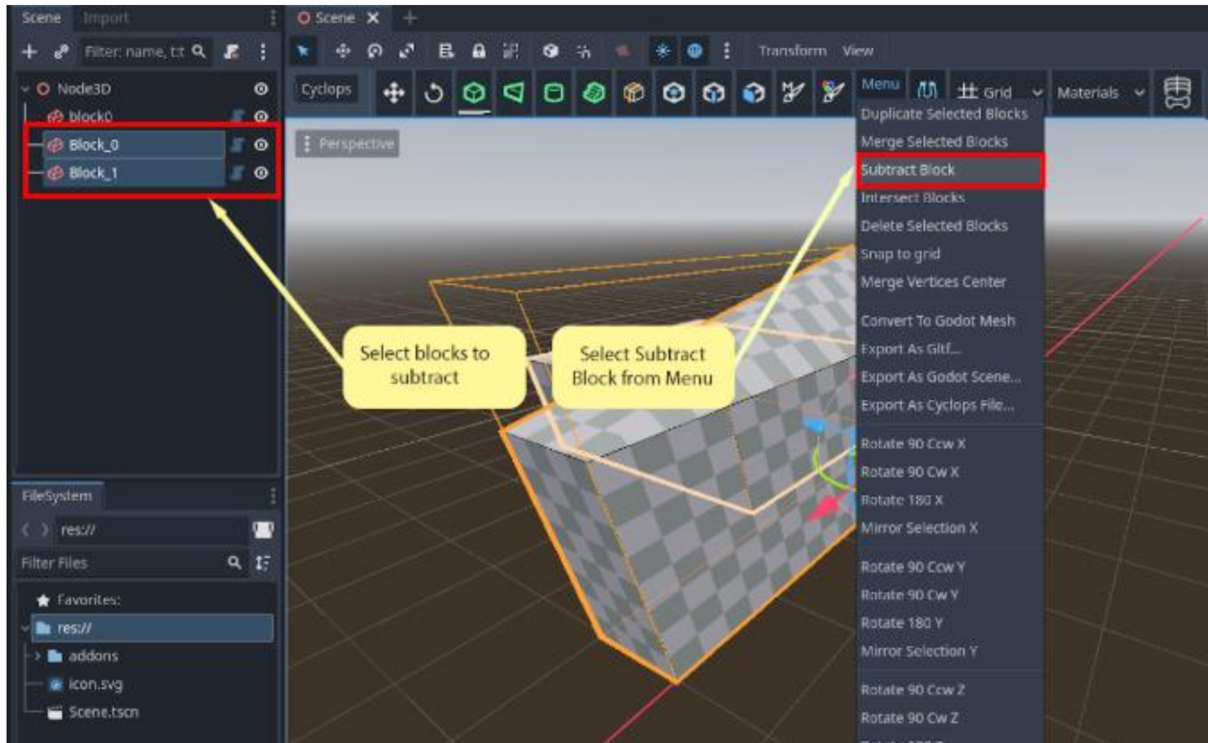

Figure 10. Example of Boolean operation in Cyclops Level Builder.

However, the Boolean operations can sometimes produce unexpected results, such as not actually performing the cut. These operations split the cut block into smaller blocks (see Figure 11). After performing multiple Boolean cuts, the number of blocks in the scene can increase significantly, making managing blocks more troublesome. Furthermore, users can merge these blocks into a single shape using the "Merge Selected Blocks" function. This sometimes also produces unpredictable outcomes, such as filling in already made cuts.

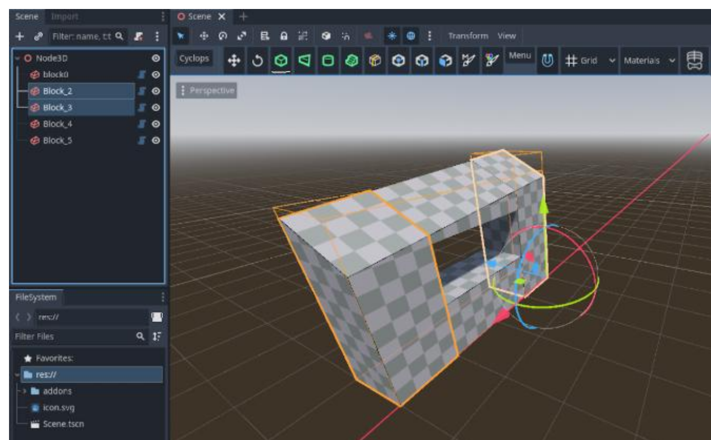
Figure 11. Result of the Boolean operation in Cyclops Level Builder.

4.2 Grid Modeler

Grid Modeler is a paid add-on for Blender. It aims to simplify hard surface modeling using a grid-based system. This makes Grid Modeler ideal for greyboxing, rapid prototyping, and creating non-organic models. To use the tool, users can first select a face, vertex, or edge of a *mesh* in "Edit Mode", then right-click and choose "Grid Modeler" from the context menu (see Figure 12). After this, a pop-up menu displays the available actions and their key bindings, and a grid is created on the selected face or vertex (see Figure 13). Users can then draw shapes on the grid using the line tool.

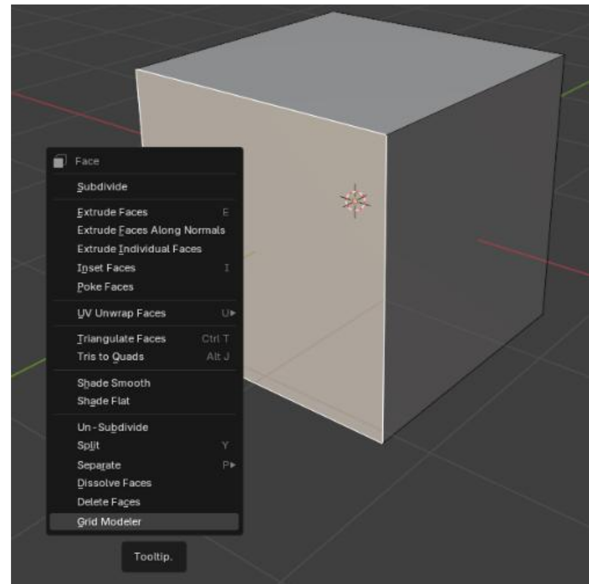

Figure 12. Accessing Grid Modeler.

Once the user has drawn their shape, they can create a new face by pressing the "3" key on the keyboard and then extruding the shape by pressing "E". The grid can also be resized by holding the "Ctrl" key and scrolling the mouse wheel to adjust the number of cells in the grid.

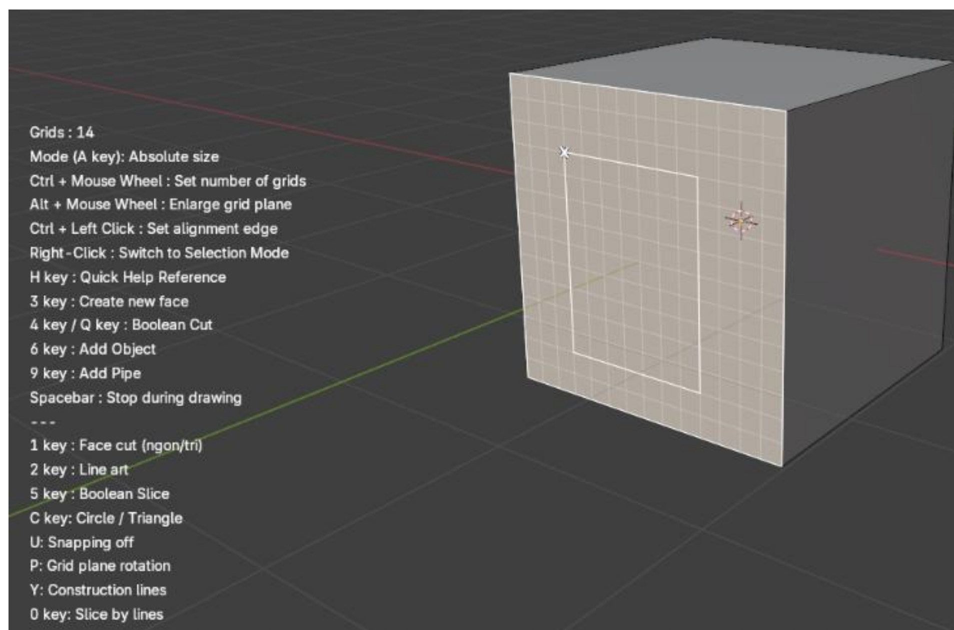

Figure 13. User interface of Grid Modeler.

For Boolean operations, the user can draw a shape with the line tool, but then press the "4" or "Q" key on the keyboard, after which a window will open where they can adjust the depth of the cut, allowing them to subtract the shape from the original mesh (see Figure 14).

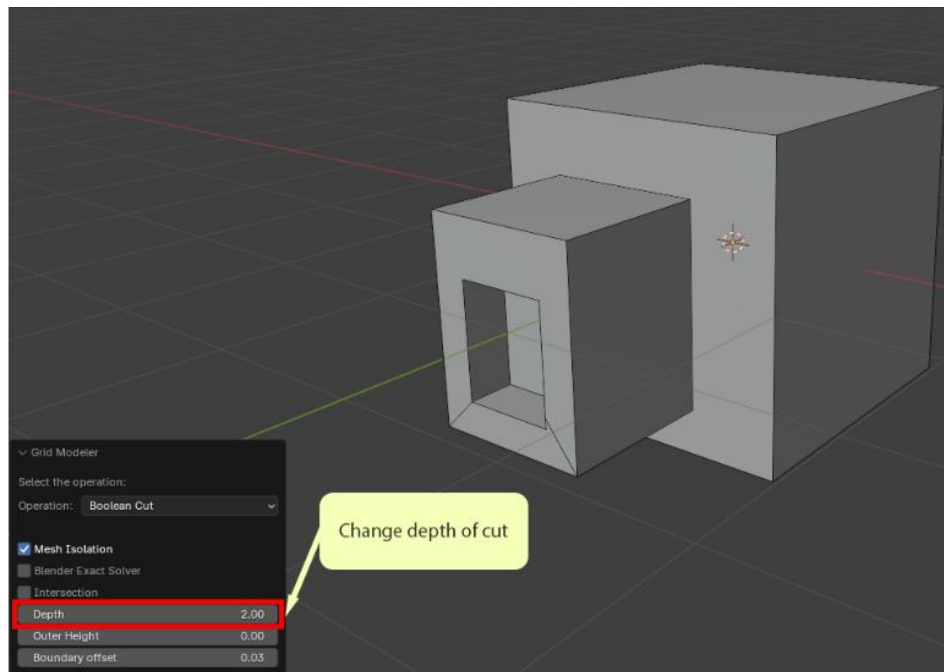

Figure 14. Cutting operation in Cube Grid.

One of the main advantages of Grid Modeler is its integration into the Blender workflow. It can be used alongside Blender's built-in tools. Using Grid Modeler, users can quickly create a basic structure, such as a building with walls and floors. Once the basic shape of the building is made, they can switch to another native Blender tool, such as Sculpting, to improve the design and add more detail. This combination of tools allows users to rapidly prototype the basic shape of the object and move seamlessly into more advanced detailing, such as smoothing, organic shaping, or adding textures.

Although Grid Modeler is a great tool, it does have some drawbacks. Users who are more accustomed to visual interfaces, such as buttons, may find the keyboard-based command menu less intuitive. Additionally, the grid's plane movement is limited, allowing it to be placed only at predefined positions, such as the faces and edges of the object. However, it is an excellent tool for users focused on hard surface modeling and rapid geometry creation.

4.3 Cube Grid

The Cube Grid tool is a native tool in Unreal Engine. It is designed for rapid prototyping, allowing users to quickly create 3D structures by adding and manipulating cubes that snap to a 3D grid. Users can adjust the grid size, which automatically resizes the cubes to match the cell size of the grid. Using simple "Push" and "Pull" commands, cubes can be extruded to create objects such as walls and ramps. Cube Grid has a user-friendly interface with all the actions displayed as clickable buttons, such as "Pull", "Push", "CornerMode" and "Flip". Additionally, keyboard shortcuts are available to quicken the process (see Figure 15).

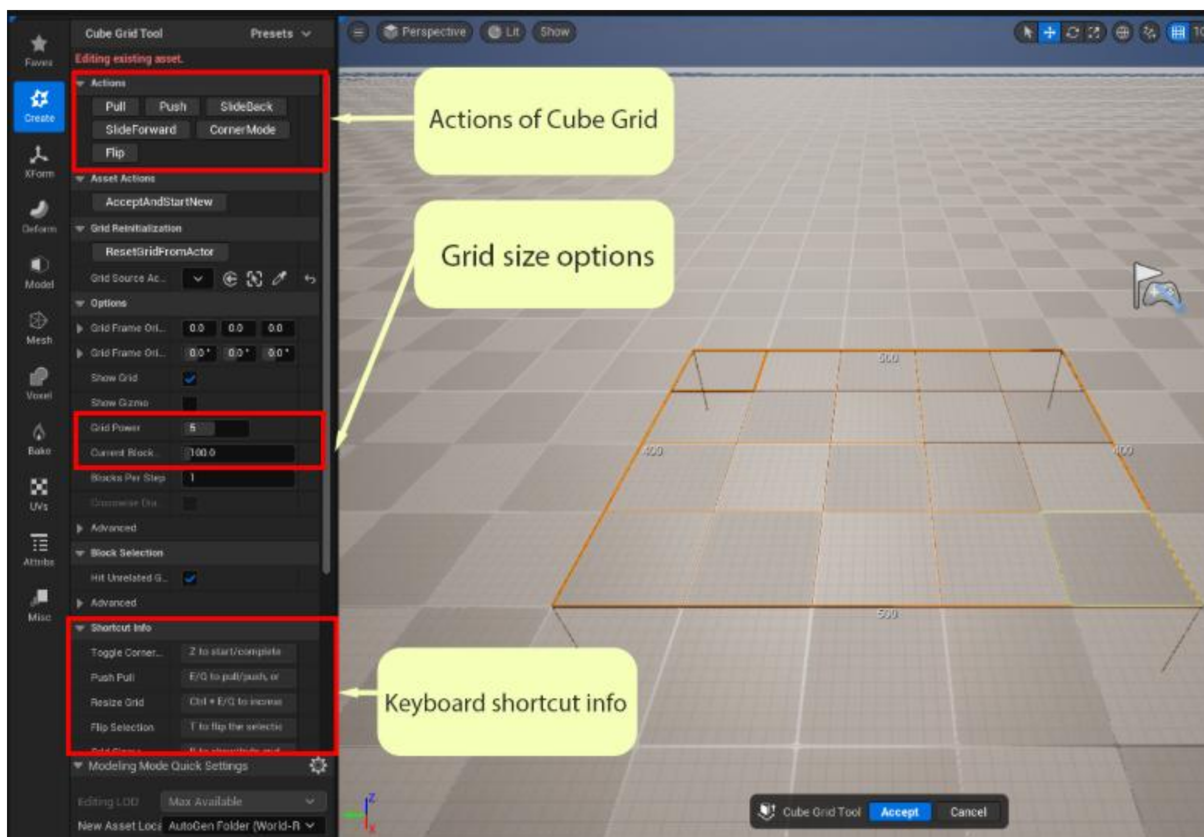

Figure 15. User interface of Cube Grid.

To begin the construction, users need to drag out a rectangle on the grid and use the "Pull" function to extrude it. An example of an extruded construct is shown in Figure 16.

Once extruded, the original construct can be modified using the "Push" function. To do this, the user first defines the shape they want to cut out from the original shape and then uses the "Push" function to cut out the defined shape. This simply just removes the block at that position; an example of a cut block is shown in Figure 17. Users can edit their shapes further by selecting "CornerMode" under the actions section. This allows them to manipulate the corners

of the shape using the "Push" and "Pull" commands, which can be used to create slopes or ramps. An example of "CornerMode" is shown in Figure 18.

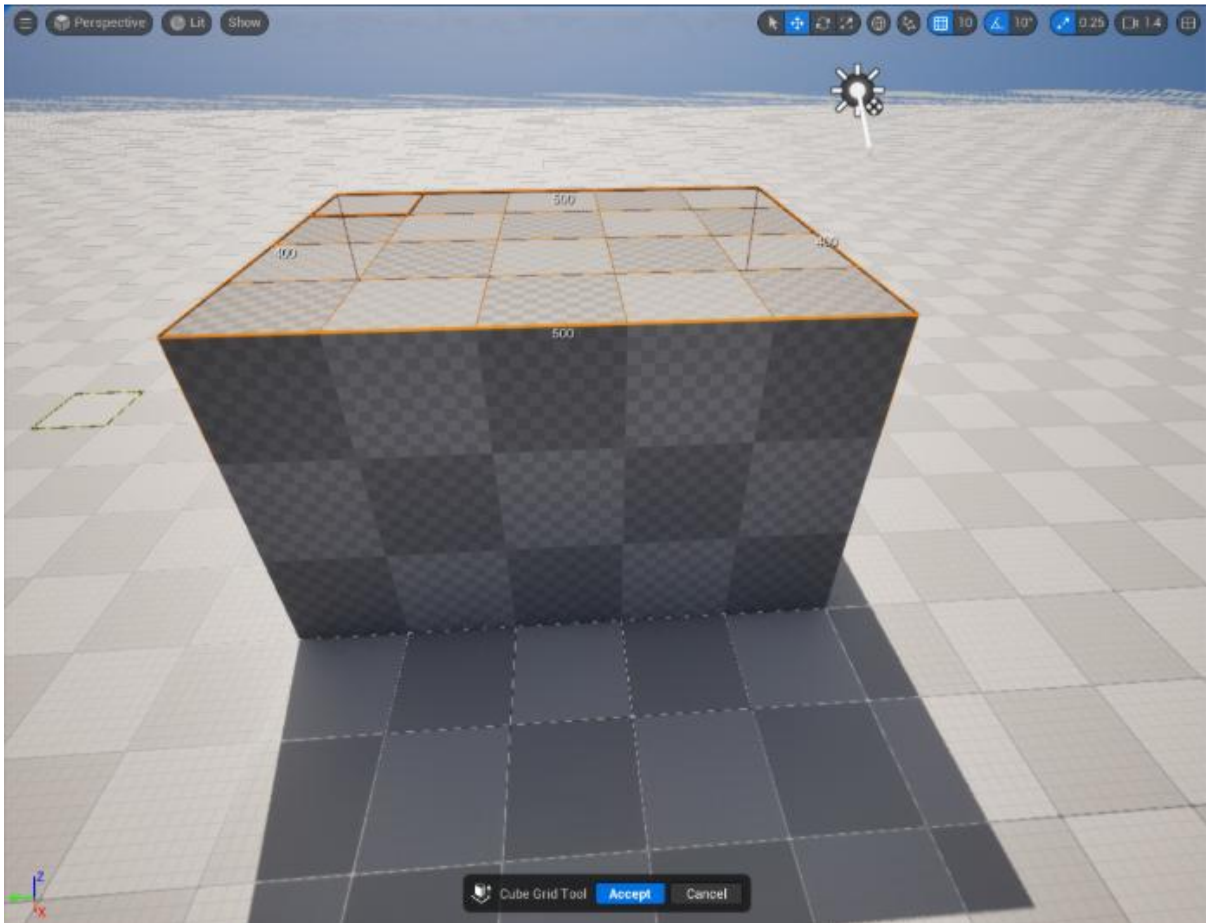

Figure 16. Example of an extruded shape in Cube Grid.

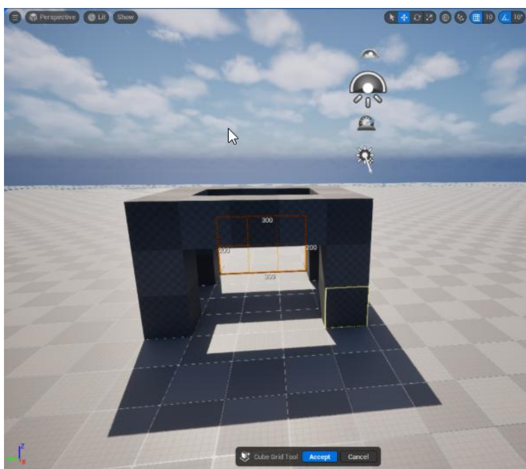

Figure 17. Example of holes created in Cube Grid.

Figure 18. Example of a slope created in Cube Grid.

The main strength of Cube Grid is its simplicity and speed, making it an ideal tool for early-level design and rapid prototyping. This tool lets users quickly create block-based structures, making it perfect for testing gameplay mechanics or creating basic environments before doing more detailed work.

Cube Grid does have its limitations. It focuses on basic cube creation, making it less suitable for advanced structures such as polygons, organic models (e.g., trees or animals), or curved objects. For these more complex models, users need to use different tools, such as Blender or additional Unreal Engine tools, like the "Extrude Polygon" tool, which allows users to draw a polygon on the grid using the line tool and then extrude it. These tools can help create more detailed and complex models, but for more organic or detailed models, it is better to use software best suited for such tasks as Blender.

5. User Experience

To activate Box Constructor, users can download it from the Godot Asset Library[14] or, if using GitHub, clone the repository. GitHub users, after cloning, need to drag the "addons" folder from the downloaded repository into the root directory of their Godot project.

Next, users must enable the plugin within Godot by navigating to Project → Project Settings → Plugins, and then checking the "Enabled" box.

Once activated, users have to right-click on `Node3D` → Add Child Node → `CubeGrid3D` to add the necessary node to the Godot scene.

To start building, the user needs to select "Move Mode" from the Godot toolbar or press "W" on the keyboard. When the `CubeGrid3D` node is selected from the Scene Tree panel, a toolbar will appear in the 3D editor's viewport (see Figure 19).

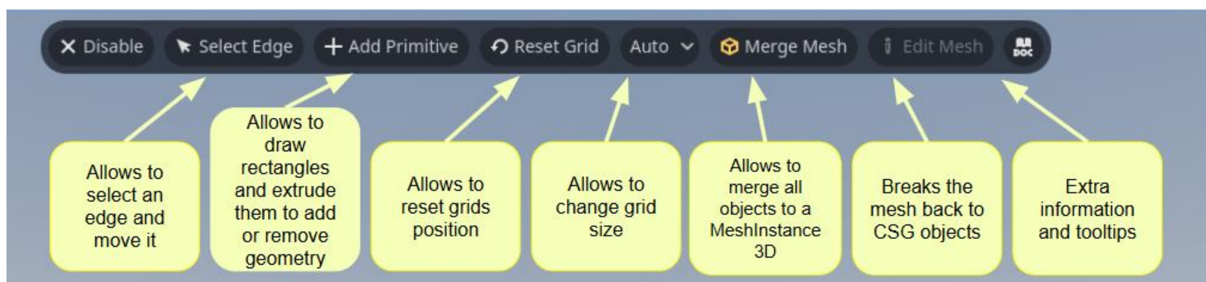

Figure 19. Box Constructor toolbar.

The toolbar has buttons for creating, modifying and optimising created geometry. Additionally, the toolbar is only visible when the `CubeGrid3D` node is selected from the Scene Tree panel. This ensures that users can keep the plugin enabled and that it will not interfere with other tools. While this subchapter briefly overviews what each button does, a more detailed explanation of the functionality and use is discussed in the next chapter.

The toolbar changes size and the visibility of buttons based on the actions the user has taken. At startup, the "Select Edge" and "Merge Mesh" buttons are disabled (see Figure 20) since the user has not created any geometry yet.

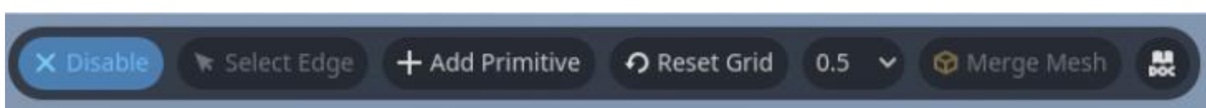

Figure 20. Box Constructor toolbar when no geometry in scene.

[14] https://godotengine.org/asset-library/asset/3944

The "Select Edge" and "Merge Mesh" buttons become available once a primitive has been added to the scene. After the user has finished creating their desired structure, they can press the "Merge Mesh" button, which merges all the newly created objects into a single `MeshInstance3D`, optimising the performance for testing. The toolbar then changes accordingly, displaying a new button called "Edit Mesh" (see Figure 21). This button allows the user to convert their "Merged Mesh" back into individual cubes, which in turn re-enables the "Add Primitive" and "Select Edge" buttons. Now the user can continue editing the primitives as before.

Figure 21. Box Constructor toolbar when the mesh has been merged.

5.1 Design of the Toolbar

The UI is an essential part of any software, as it is the primary method for users to interact with its functionality. Box Constructors aimed to create a simple, user-friendly interface that guides the user through the level creation process. The design choices were influenced by Theo Mandel's "Golden Rules"[4], which are as follows:

1. Putting the user in control – users should be able to choose their preferred input method, whether keyboard or mouse, allowing for flexibility and accessibility.
2. Minimising cognitive load – through intuitive design and feedback, reducing the mental effort required to use the system.
3. Ensuring consistency – maintaining uniformity of design in the software.

For Box Constructor, cognitive load was reduced by giving the user feedback by highlighting the currently selected button. Additionally, irrelevant buttons are either disabled or hidden to prevent unnecessary distractions and unintended actions, such as attempting to draw while a mesh is merged. Furthermore, a tooltip dropdown was added to each button. When the user hovers over a button, a short description is given of what that button does, along with a corresponding keyboard shortcut. These tooltips reduce the need for users to reference the user manual. Finally, consistency was achieved through the uniform design of the toolbar. All toolbar buttons share a cohesive look but are uniquely labelled and paired with an icon so that users can quickly distinguish them.

6. Implementation

This chapter provides an overview of the development process and features of Box Constructor. The tool uses CSG nodes, allowing users to place `CSGBox3D's` quickly and efficiently. Operations for the CSG are also set automatically depending on the extrusion direction. This eliminates the need for users to manually configure each node's position and operation, streamlining the workflow.

The chapter is structured as follows:

1. Chapter 7.1 discusses the engine version used.
2. Chapter 7.2 explains the grid system and the snapping functionality.
3. Chapter 7.3 explains the geometry creation workflow.
4. Chapter 7.4 details the edge manipulation method.
5. Chapter 7.5 explains the mesh merging and reconstruction system.

6.1 Setup and Architecture

Godot was chosen as the game engine for Box Constructor because of its strong community support, extensive documentation, and open-source nature. Its Asset Library makes it easy for developers to create and distribute plugins. Additionally, GDScript, Godot's built-in scripting language, is simple to learn and use. Creating custom tools in Godot is also easy; developers only need to add the `@tool` annotation at the top of the script to enable it to run in the editor.

Box Constructor was developed using Godot Engine version 4.4, the latest stable release at the time of development. This version was chosen because it introduced improvements to the CSG system, such as fixing mesh corruption issues in large or complex models and improving overall stability[15]. These updates were useful as CSG functionality is at the plugin's core.

The CSG operations supported in Godot are:

- Union – the geometry of both primitives is merged, intersecting geometry is removed.
- Intersection – only intersecting geometry remains, the rest is removed.
- Subtraction – the second shape is subtracted from the first, leaving a dent with its shape.

[15] https://godotengine.org/releases/4.4/

6.2 Grid and Snapping

The grid was implemented by creating a custom node called `CubeGrid3D`, which consists of a `PlaneMesh` and `BoxCollisionShape`, scaled to cover a large area. The `PlaneMesh` is assigned a custom shader material that renders the grid lines onto it. The grid's size can be adjusted manually using the predefined values selected from the drop-down menu in the toolbar. When the user chooses a value, the `_on_grid_size_changed()` function updates the `grid_scale` parameter in the shader code, changing the grid's scale accordingly. Figure 22 illustrates the grid pattern at scale 5, while Figure 23 shows the grid at scale 1.

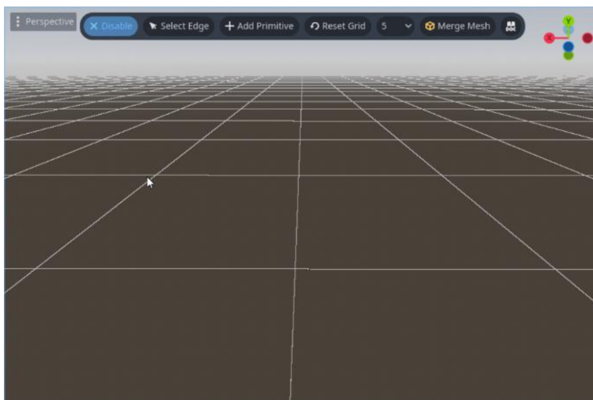

Figure 22. Grid scale at 5.

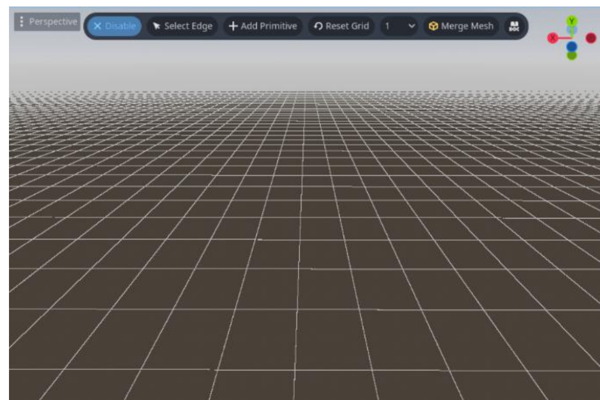

Figure 23. Grid scale at 1.

In addition to resizing, the grid can be repositioned, allowing users to place it on any surface of the created geometry (see Figure 24). When the user presses the "X" key on the keyboard, the function `_align_grid_to_surface()` casts out a ray from the mouse, which detects the hit position and the *surface normal* of the hit surface. The `CubeGrid3D's PlaneMesh` and `CollisionShape` are then moved and rotated to align with the surface the mouse was hovering over.

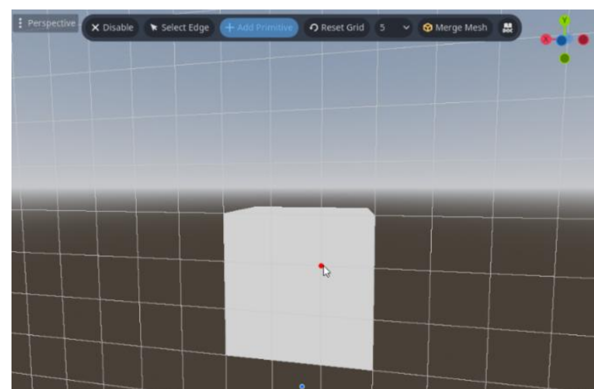

Figure 24. Grid placed on the vertical surface of a cube.

The snapping functionality ensures that the created geometry aligns precisely with the grid. The `_snap_to_grid()` function takes a `Vector3` position as input and adjusts it to the nearest grid point. It works by dividing the clicked position by the grid scale to calculate its relative position on the grid. This relative position is then rounded to the nearest integer, snapping it to

the closest grid cell. Finally, the result is multiplied by the grid scale to convert it back to world coordinates.

6.3 Geometry Creation

The geometry creation process in Box Constructor can be divided into three steps:

1. the user defines the size of the base rectangle;
2. the user extrudes the rectangle into a 3D shape;
3. the user confirms the extrusion and its operation.

The following subchapters discuss the process and methods used to create 3D shapes in Box Constructor.

6.3.1 Base Rectangle Drawing

The first step is to define the base rectangle. When the user presses the "Add Primitive" button in the toolbar, a red dot appears on the surface where the mouse is hovered. This red dot marks the starting point for drawing, showing the current mouse position within the grid system, snapping to the nearest grid point (see Figure 25).

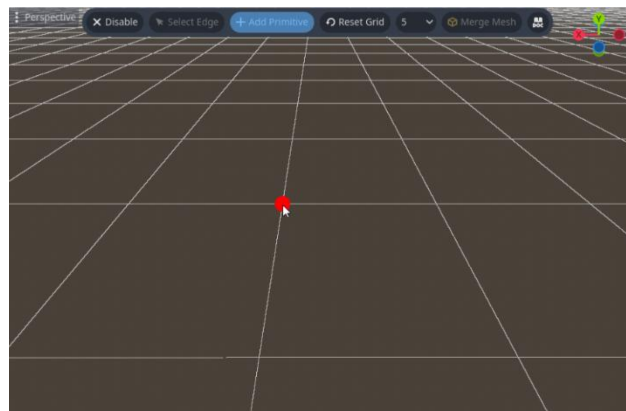

Figure 25. Indicator of mouse position in the grid.

When the user presses the left mouse button, a ray is cast from the camera through the mouse position into the 3D world to detect the surface on which the rectangle will be drawn. If a surface is hit, the surface normal is saved to define the rectangle's orientation. The point of intersection is then snapped to the grid and set as the starting point (`draw_start`). A drawing plane is then created using the `draw_start` position and normal. All further ray intersections are performed relative to the drawing plane. Initially, the rectangles' start and endpoint are the same, but as the user drags the mouse, the rectangles' endpoint (`draw_end`) is updated. The rectangle's corners are recalculated each time the user moves their mouse using the `_calculate_base_rect_points()` function.

The `_calculate_base_rect_points()` function takes the two points `draw_start` and `draw_end` and calculates the two remaining corners by combining the two endpoints' X, Y and Z values. The result is a rectangle that fits within the area between these two points, returning the rectangle's bounding box (see Figure 26).

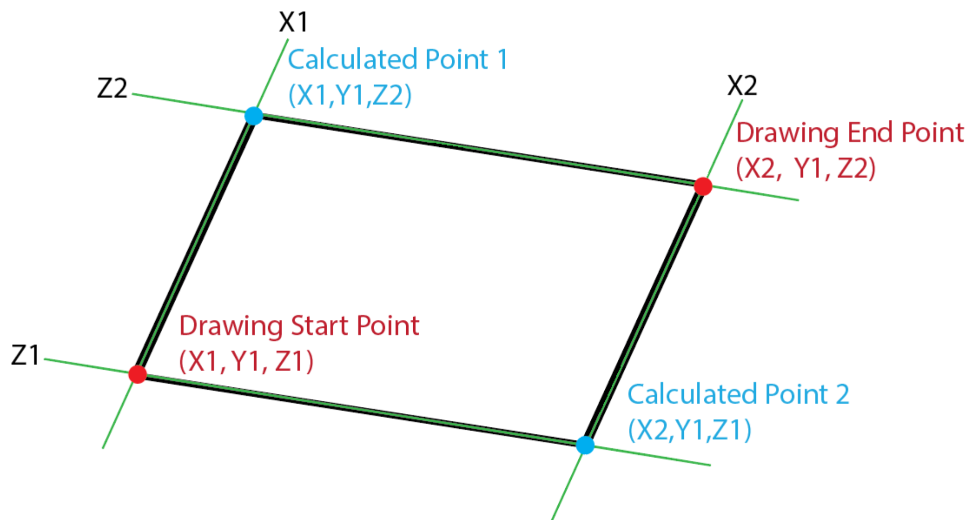

Figure 26. Base rectangle calculation.

To visualise the rectangle, `MeshInstance3D` is used. The `_create_rectangle_preview()` function creates the mesh and its material, while `_update_rectangle_preview()` keeps updating this mesh while the user drags the rectangle. The lines are drawn using the `_add_thick_line()` function, which takes two points and draws a line between them using a rectangle created by two triangles. When the user releases the mouse, the base rectangle is finalised, and the final dimensions and orientation of the rectangle are saved (see Figure 27).

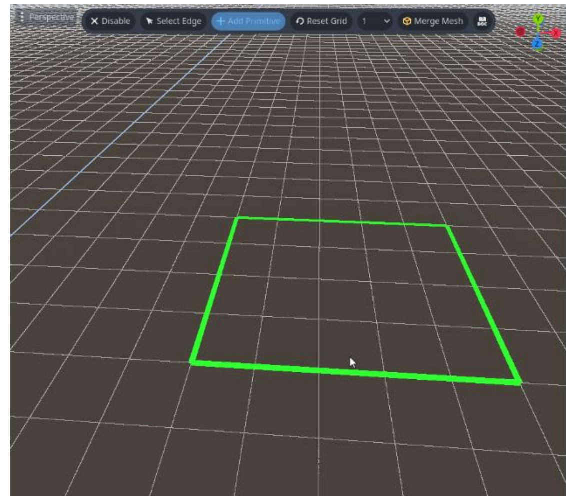

Figure 27. Base rectangle preview in Box Constructor.

6.3.2 Extruding

After defining the base rectangle, the user can extrude it into a 3D shape by moving their mouse to specify the extrusion height. The direction of the extrude is determined by the `extrude_line_normal` variable, which was stored when the base rectangle was created.

An invisible line is created along the extrusion normal, starting at the initial extrusion point, the last place where the mouse was released during the drawing of the base rectangle. This line extends 5000 units in both directions. The endpoints of the line are marked as Extrude Endpoint 1 and Extrude Endpoint 2 (see Figure 28). As the user moves their mouse, a ray is cast from the camera through the current mouse position, marked as Mouse Ray. The closest point on the

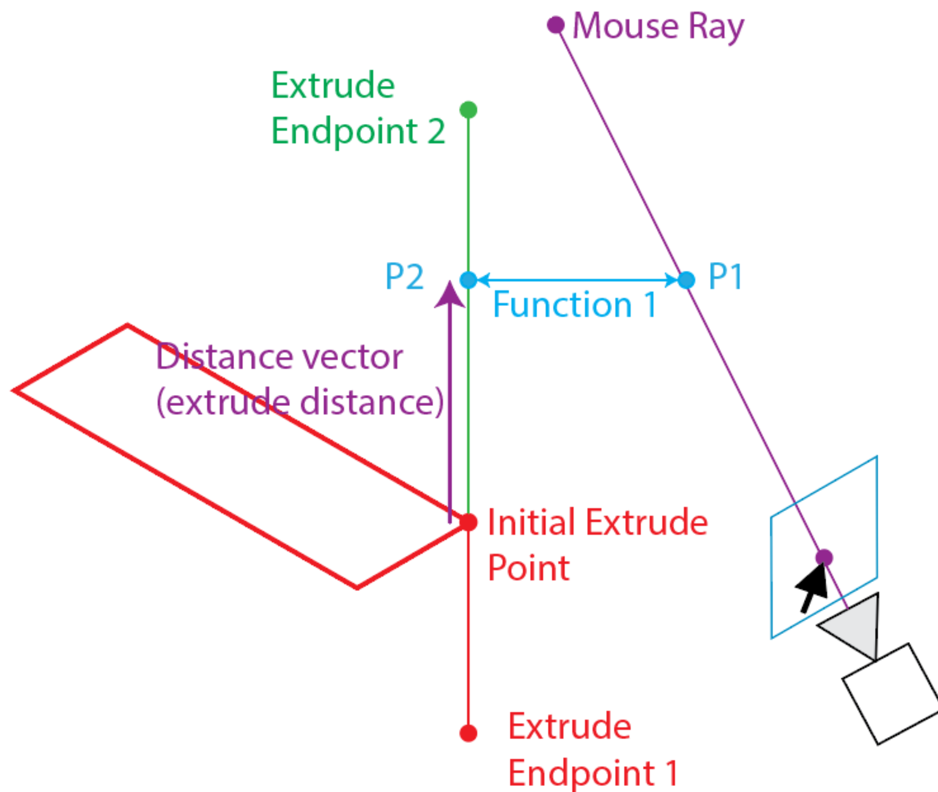

Figure 28. Extruding logic in Box Constructor.

extrude line is determined using the `Geometry3D.get_closest_points_between_segments()` function marked as Function 1, which calculates and returns the two points with the shortest distance between them. The closest point marked as P1 is the extrusion height. Hence, the extrusion distance is calculated as the vector length between the initial extrude point and the nearest point on the extrusion line, in this case, P1. This distance is signed using the dot product with the extrusion normal, and the result is snapped to the grid to ensure that the extrusion is in the correct direction. Taking the dot product ensures the extrusion is in the correct direction. A positive result means that it aligns with the normal, while a negative result indicates the opposite direction. Once the user clicks again, the extrusion is confirmed, the rectangle is extruded into a finalised 3D shape, and the corresponding sized `CSGBox3D` is created.

The extrusion preview also changes colour depending on the extrusion direction.

- If the extrusion is in the positive direction, meaning it aligns with the extrusion normal, the preview outline turns green (see Figure 29), symbolising the addition of new geometry. The result can be seen in Figure 30.
- If the extrusion is in the negative direction, meaning it is in the opposite direction of the extrusion normal, the preview outline turns red (see Figure 31), indicating that geometry will be removed. The result can be seen in Figure 32.

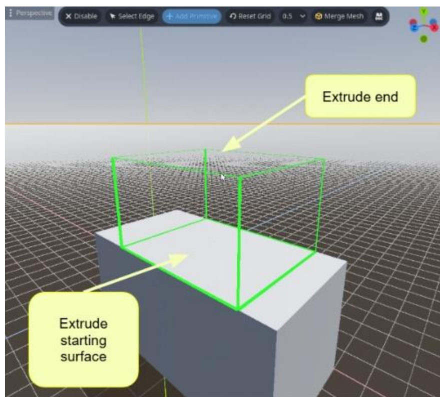

Figure 29. Extrusion in the positive direction.

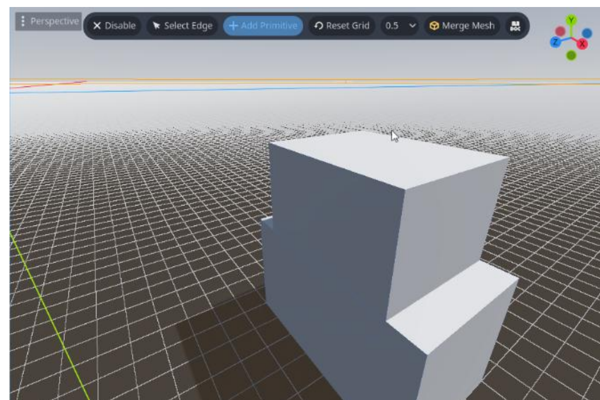

Figure 30. Result of the extrusion in the positive direction.

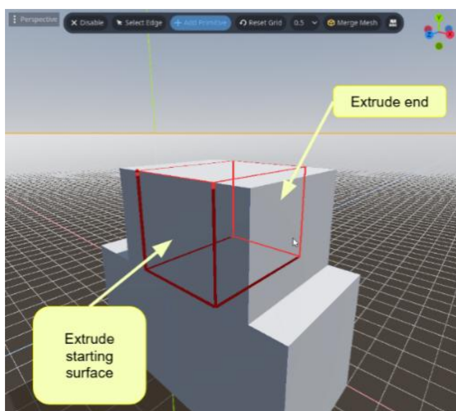

Figure 31. Extrusion in the negative direction.

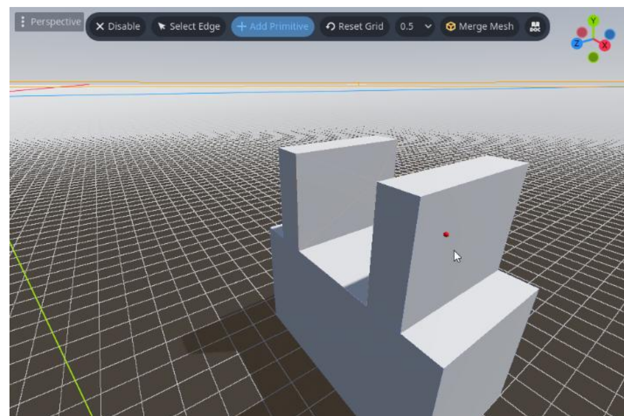

Figure 32. Result of the extrusion in the negative direction.

6.4 Corner Movement

When the user selects the "Select Edge" tool from the toolbar, the closest edge to the mouse is highlighted with a red preview line. Edge detection is implemented in the same way as the extrusion logic. A ray is cast from the camera through the mouse position into the 3D world, and the system calculates the closest points between the mouse ray and each edge segment using the `Geometry3D.get_closest_points_between_segments()` function. The distance between the mouse ray and each edge is determined, and the edge with the shortest distance is highlighted (see Figure 33).

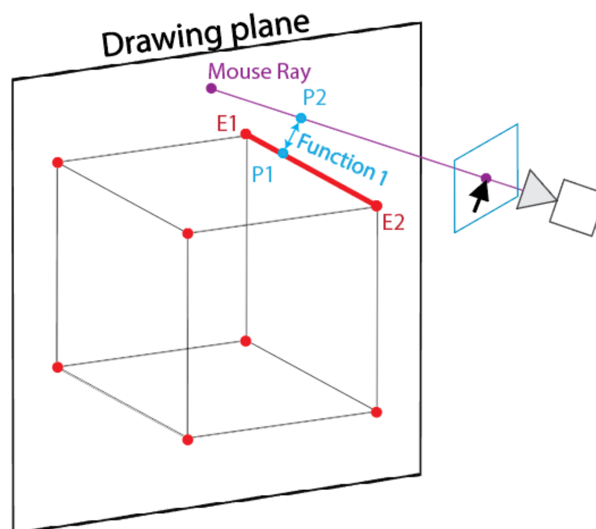

Figure 33. Edge detection logic in Box Constructor.

Once an edge is highlighted (see Figure 34), and the user clicks the left mouse button, a plane is created for dragging. This plane is perpendicular to the edge and is used to constrain the movement of the edge during the drag operation. If the selected object is a `CSGBox3D`, it is first converted into a `CSGMesh3D` to enable vertex manipulation. This conversion is necessary because `CSGBox3D` nodes are procedural shapes that do not expose their vertices for direct editing, whereas `CSGMesh3D` nodes allow adjusting individual vertices. This is done using the `_convert_box_to_CSGMesh()` function, which takes the size of the `CSGBox3D` and calculates all its eight vertices. After calculating the vertices, faces are created by defining the indices that connect these vertices to form triangles. Two triangles represent each face of the box. The vertices and indices are then added to an `ArrayMesh` using the `add_surface_from_arrays()` method. This `ArrayMesh` is assigned to a new `CSGMesh3D` node, which replaces the original `CSGBox3D` node. The mouse movement is projected onto the plane during the drag operation, and the resulting position is snapped to the grid. The two vertices corresponding to the selected edge

are adjusted by applying an offset based on the new snapped mouse position. When the user clicks the mouse button, the dragging operation stops, the preview line is hidden, and the mesh remains updated, reflecting the changes made during the drag operation (see Figure 35).

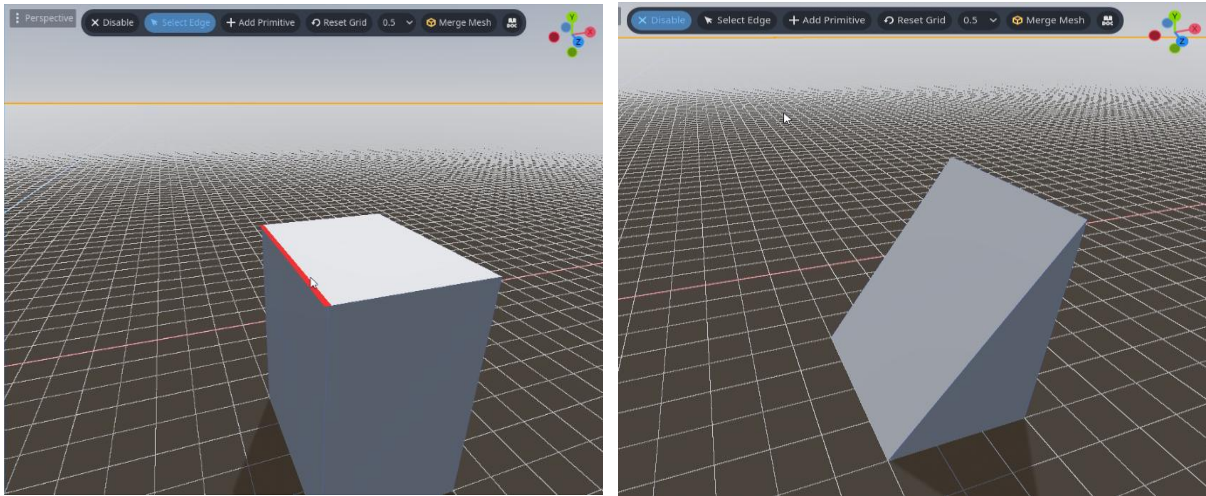

Figure 34. Edge highlight in Box Constructor.

Figure 35. Slope created in Box Constructor using "Select Edge" tool.

6.5 Mesh Merging and Reconstruction

Box Constructor also has a mesh merging and reconstruction feature that allows users to optimise their created 3D models for performance while allowing them to edit them later. This is done because the CSG nodes are intended only for level prototyping. Creating CSG nodes has a significant CPU cost compared to creating a `MeshInstance3D`[16]. The merging process combines the CSG nodes into a single `MeshInstance3D`, while the reconstruction process restores the original CSG nodes from the merged mesh.

6.5.1 Merging

The mesh merging process starts when the user presses the "Merge Mesh" button in the toolbar, which calls the `_on_merge_mesh()` function that checks if the `csg_root` has any CSG nodes as children or a node called "MergedMesh" as a child.

The merging process iterates over all the child nodes of the `csg_root`, only picking out `CSGBox3D` and `CSGMesh3D` nodes. Subtractive CSG nodes first go through an intersection check to determine if they remove any geometry. If the subtractive node does not intersect with any

[16] https://docs.godotengine.org/en/stable/classes/class_csgbox3d.html

additive nodes, it is left out of the final mesh. This optimises any unnecessary calculations in the reconstruction method and removes irrelevant geometry.

Once all the relevant nodes are identified, their information, such as position, size and operation type, is stored using the `_store_csg_data()` function, which saves all of the info into the metadata of the "MergedMesh". The remaining nodes are combined using the `CSGCombiner3D.get_meshes()` method, which performs all Boolean operations and returns the result mesh. This mesh is then stored in the `MeshInstance3D` named "MergedMesh", which is added to the scene as the child of the `csg_root`. The metadata from the original CSG nodes is stored using the `set_meta()` method, allowing it to be read when reconstructing the original CSG nodes later. The final step is the removal of all of the CSG nodes from the `csg_root`, which only leaves the “MergedMesh” as the child of the `csg_root` node.

6.5.2 Reconstruction

The reconstruction process starts when the user presses the "Edit Mesh" button in the toolbar, which calls the `_on_edit_mesh()` function that checks if the `csg_root` contains a node called “MergedMesh”. If the node isn't found, the function returns with a warning.

The reconstruction process reads in the stored metadata from the "MergedMesh” node and uses it to recreate the original CSG nodes. This is done using the `_convert_to_boxes()` function, which iterates over the metadata and creates either a new `CSGBox3D` or a `CSGMesh3D`, based on the info in the metadata. `CSGBox3D` nodes have their size and position restored. For `CSGMesh3D` nodes, the vertex and index data are used to recreate the mesh. The recreated nodes are added to the scene as children of the `csg_root`, and the node “MergedMesh” is removed. Allowing users to do CSG operations between the CSG nodes.

7. Testing

A usability test was conducted with four participants to evaluate Box Constructor. The test was structured as a practical exercise, where participants tried out three different greyboxing tools and assessed their ease of use. The task was to build a basic house using Cube Grid, Cyclops Level Builder, and Box Constructor, in the same manner for each tool. The test was carried out via Discord, where participants' actions were observed, and they provided feedback through a short questionnaire. The first part of the survey gathered background information on their experience with game engines. They answered questions like "How long have you used game engines?" and "What is your primary engine?". This gave context to how intuitive the tools were for users with varying experience levels. Participants rated each tool's usability using the Usability Metric for User Experience (UMUX), a four-statement questionnaire evaluated on a Likert scale from 1 to 7 [5]. The participants also recorded how long each task took, measuring completion time in minutes. During the test, participants received minimal assistance, and help was only provided if they had questions or got stuck on a task.

7.1 UMUX

The Usability Metric for User Experience (UMUX) is a shorter alternative to the System Usability Scale (SUS) for evaluating software usability. While SUS has 10 statements, UMUX is more compact, using only four. This makes it quicker to complete and leads to higher response rates.[17]

According to Finstad [5], UMUX correlates strongly with SUS, providing consistent and reliable usability insights. It also focuses on one usability factor, ensuring a clear and straightforward evaluation. This makes UMUX a fast, effective way to measure the usability of software.

The four statements of UMUX are:

1. this system's capabilities meet my requirements;
2. using the system is a frustrating experience;
3. the system is easy to use;
4. I have to spend too much time correcting things within the system.

[17] https://blog.uxtweak.com/umux/#:~:text=%F0%9F%93%8A%20UMUX%20is%20a%20powerful%20benchmarking%20tool%20that,that%20helps%20teams%20iterate%20and%20improve%20products%20rapidly.

The UMUX score is calculated using these statements. First, one is subtracted from the positively worded statements 1 and 3, and their results are summed. Next, for the negatively worded statements 2 and 4, their response scores are subtracted from 7. The combined results are then added and multiplied by 4.1667 (or 100/24), giving a UMUX score for a participant ranging from 0 to 100. [5]

Since the survey had four participants to get the average score for each tool, all the answers were summed up and divided by the number of participants.

7.2 Participants

The survey participants had varying experience with game engines. Most of the participants were not frequent game engine users. Of the four participants, two said they use game engines about once a month, one uses it weekly, and one once a year (see Figure 36).

When asked about their experience, most users had 0 to 6 months of experience, while one out of four participants had 1 to 2 years of experience (see Figure 37). Thus, the feedback mostly came from beginner users. When asked what engine the participants had used the most, Unity was the most popular choice (see Figure 38).

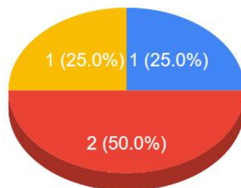

Figure 36. Participants' answers to the frequency of game engine use.

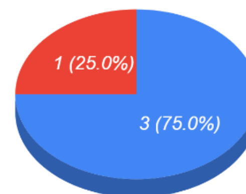

Figure 37. Participants' answers to experience with game engines.

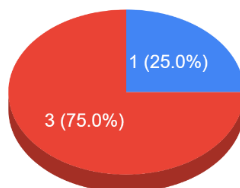

Figure 38. Participants' answers to main game engine.

7.3 Usability Evaluation

Before starting the task, all participants first watched an introductory video showcasing the functionality of Cube Grid and Cyclops Level Builder. When using Box Constructor users had to read the user manual first. It is important to note that the time it took to watch the video or look through the manual was not counted in the task completion times.

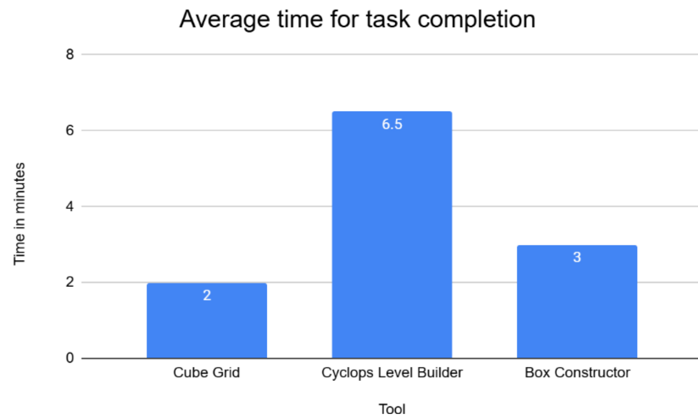

Figure 39. Average task completion in minutes.

When the users were ready, they started the timer and began the task, recording how long it took them to finish building the structure. Cube Grid took on average 2 minutes, Cyclops Level Builder 6.5 minutes, and Box Constructor 3 minutes (see Figure 39). Additionally, separate times for each participant are provided (see Figure 40).

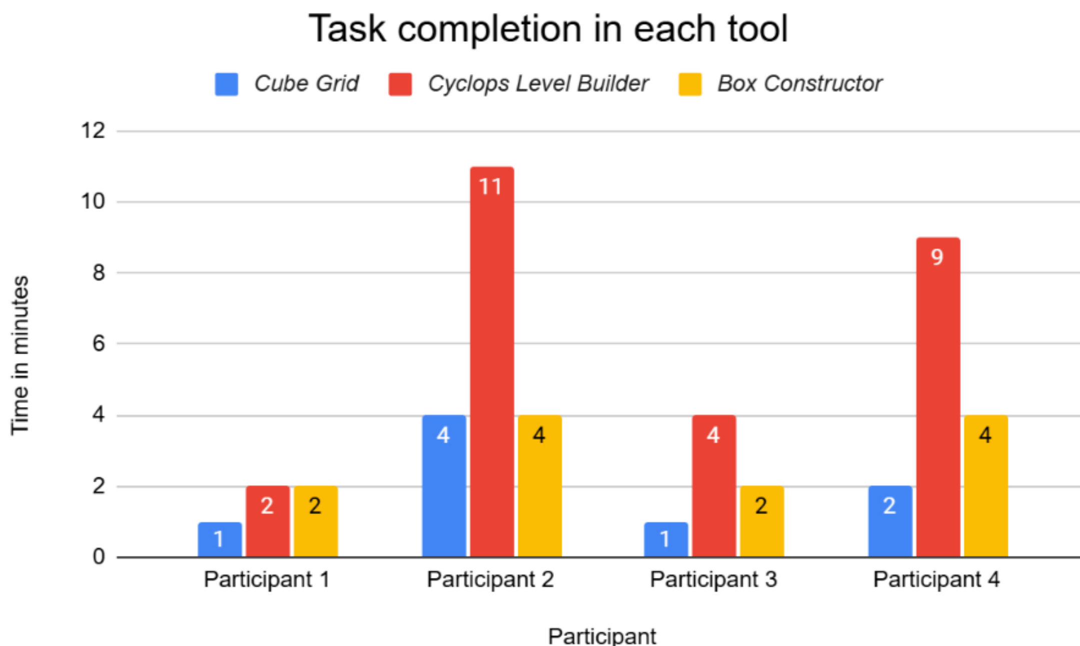

Figure 40. Task completion per participant in minutes.

None of the participants encountered any issues while using Cube Grid. Many praised the tool for its simplicity and liked the keyboard-based workflow. However, Cyclops Level Builder took longer due to various problems experienced by participants. Some struggled with Boolean operations, while others experienced crashes. One participant chose not to use Boolean operations and instead constructed holes manually by leaving gaps in the geometry. Another participant encountered a bug where the Boolean operation failed, requiring them to restart Godot, after which the issue was resolved. Box Constructor also took slightly longer than Cube Grid due to some users experiencing issues, such as the drawing starting on the wrong plane, and they had to readjust the camera to fix the problem. Additionally, one participant had difficulties distinguishing the vertices where the mouse snaps to because the created cubes are flat-shaded, and the grid lines were hard to see.

The UMUX score for each tool was calculated (see Figure 41). Cube Grid achieved the highest usability score with an average score of 93.8 out of 100. Cyclops Level Builder got an average score of 52.1 out of 100. Box Constructor got an average score of 80.2 out of 100. Additionally, the individual UMUX scores of each participant are included (see Figure 42).

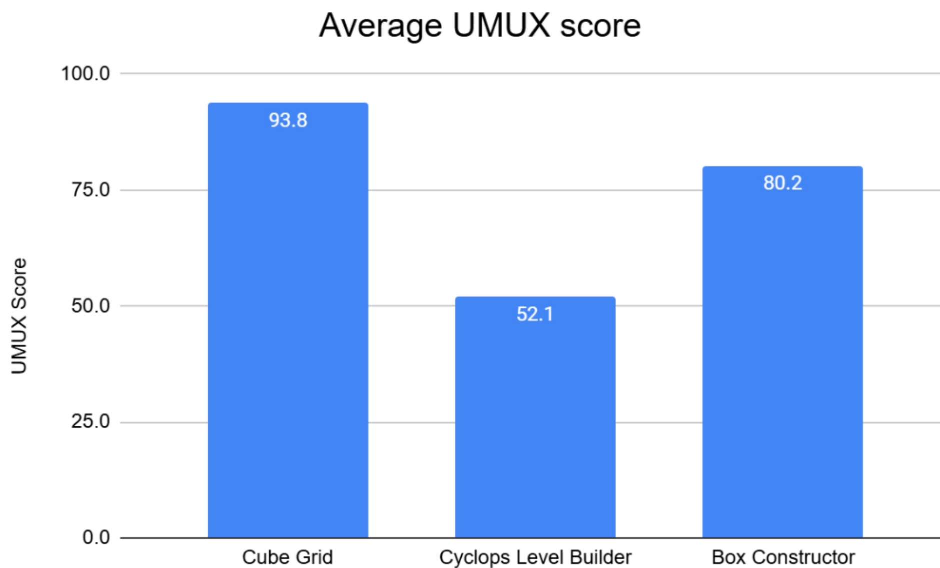

Figure 41. Average UMUX score of each tool.

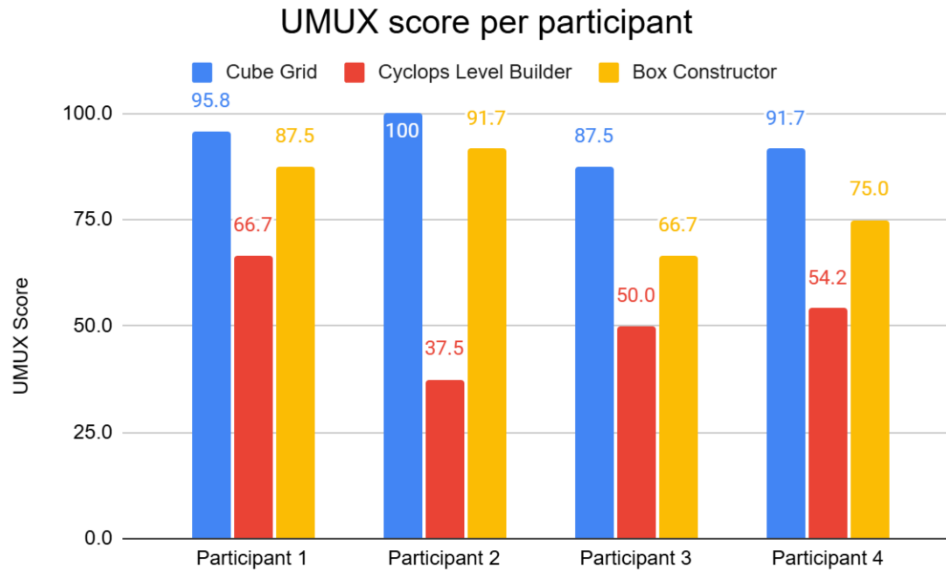

Figure 42. Calculated UMUX score based on the participants' answers.

7.4 Overall Experience and Preferences

The overall feedback on Box Constructor was positive, but users noted several missing features that could enhance its usability. One of the most frequently suggested improvements was the addition of an undo function (Ctrl + z), as participants had to manually delete blocks when they made a mistake. Users identified that selecting and deleting blocks is an issue. To address this, they suggested adding a selection tool to allow them to click on a block and press delete to remove it from the `Scene Tree`. Participants also recommended expanding the shape options, such as adding stairs like in Cyclops Level Builder. These ideas are great, as incorporating more shapes would allow users to create more detailed prototypes that better reflect the actual map design. This concept could be expanded by enabling users to draw custom shapes using a line tool and then extrude them into 3D shapes, like in Grid Modeler. When asked which tool they preferred for greyboxing, all participants chose Cube Grid, due to its simplicity and ease of use.

To enhance the usability of Box Constructor, several additional features could be implemented inspired by Cube Grid:

- Keyboard-based extrusion – users liked this feature in Cube Grid, so Box Constructor could introduce a similar option where shapes could be extruded using keyboard shortcuts, giving users more flexibility to choose their preferred method.

- Improved cell highlighting – instead of a simple dot indicator, the highlight could cover the entire cell, similar to Cube Grid, making it more intuitive for users to understand which cell they are hovering over.
- Numeric dimension indicators – adding numbers on the lines to show the base rectangle dimensions and extrude height gives users an idea of how many blocks they add or remove.
- An undo option – users could easily correct their mistakes instead of manually deleting blocks.

7.5 Performance Testing

Different tool operations were tested using a script that generated numerous 1x1 cubes. All these cubes were set to perform the `UNION` operation and were arranged to touch each other, creating a large structure. This was done to mirror typical usage of the tool for constructing structures made up of many interconnected cubes. This method ensures that the CSG operations are executed in a way that reflects real-world usage scenarios.

The performance tests were conducted on a computer with the following specifications: AMD Ryzen 9 5900HS, NVIDIA GeForce RTX 3060 Laptop GPU with 6 GB GDDR6 memory, and 32 GB DDR4 RAM.

The following operations were tested:

- Adding a new cube – the timer started the moment the extrusion was defined and stopped when the `CSGBox3D` was created with the operation type `UNION`.
- Cutting a cube – the timer was started the moment the extrusion was defined and stopped when the new `CSGBox3D` was created with the operation type `SUBTRACT`.
- Edge movement – the timer was started when the edge was selected and stopped once the edge snapped to the new mouse position.
- Merging – the timer was started when the "Merge Mesh" button was clicked and stopped when the operation completed.
- Editing – the timer was started when the "Edit Mesh" button was clicked and stopped when the operation completed.

Each operation was timed using the `Timer` class to track execution time across varying cube counts. The testing was conducted in stages. First, 500 cubes were added to the scene, and then different operations were tested.

For the addition operation, a new cube was created, and the time was measured to determine how long it took to complete the operation. Next, the cube was cut by a separate block, and the execution time was re-recorded. Once all operations had been measured, the blocks were deleted. This process was repeated with block counts of 1000, 3000, 5000, 7000 and 9000 in the scene. At lower values, such as 500 or 1000 cubes, the execution time was nearly instantaneous. However, performance started to decline at around 3000 cubes (see Figure 43). At 7000 cubes, the Godot engine began to experience significant slowdowns and performing actions in the editor, such as selecting cubes from the `Scene Tree` and trying to delete, took time. By 9000 cubes, performance degraded even further, occasionally leading to crashes because Godot has trouble handling the number of `CSGBoxes` in the scene. The results of the performance test can be seen in Table 1. The column "cubes in the scene" represents the number of blocks added, while the operation columns indicate execution time in seconds.

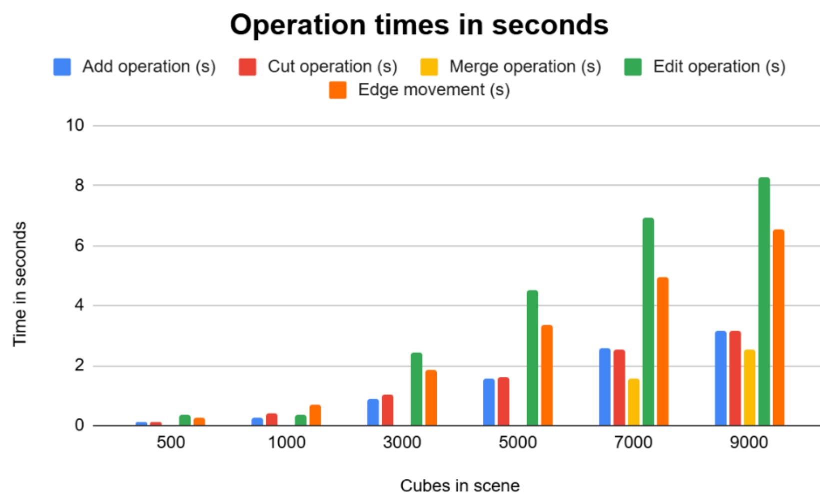

Figure 43. Results of performance testing.

Table 1. Performance testing results in seconds.

Cubes in scene	Add operation	Cut operation	Merge operation	Edit operation	Edge movement
500	0.1	0.1	0.0	0.4	0.3
1000	0.3	0.4	0.0	0.4	0.7
3000	0.9	1.0	0.0	2.4	1.9
5000	1.6	1.6	0.0	4.5	3.3
7000	2.6	2.5	1.5	6.9	5.0
9000	3.2	3.2	2.5	8.3	6.5

8. Conclusion

The goal of this thesis was to develop a tool that enables the intuitive creation of 3D objects in the Godot game engine. Box Constructor allows rapid construction of simple level layouts using the greyboxing method. The tool addresses the absence of an accessible and user-friendly solution for the Godot engine.

The thesis gave an overview of level design and its principles, presenting a structured approach by the authors of *Practical Game Design* [2]. The thesis examined various prototyping methods, including paper and digital prototyping and discussed their benefits and drawbacks. Several popular greyboxing tools were also analysed, such as Cyclops Level Builder for Godot, Grid Modeler for Blender, and Cube Grid for Unreal Engine. Their functionalities and workflows were explored to identify features that could enhance the usability and functionality of Box Constructor.

An overview of the user interface of Box Constructor was provided, along with best practices for user interfaces inspired by Theo Mandel's "Golden Rules" and the implementation details of the tool. Box Constructor allows users to quickly add and remove new geometry by changing the extrusion direction. When the user changes the grid scale, the size of the created cubes also adjusts accordingly. The grid can be moved to any surface of a block. Additionally, users can manipulate the edges of cubes to construct diagonal slopes.

The testing methodology was explained, and the results were presented. Box Constructor successfully addressed Godot's lack of accessible greyboxing tools, achieving an 80% usability score in testing. These results demonstrated strong usability, although certain areas were identified for improvement. Future improvements were proposed based on the user feedback and comparisons with other tools, including refining the drawing system to ensure that it starts on the correct plane, improving cell highlighting, introducing keyboard-based extrusion inspired by Cube Grid, and adding functionality for undoing actions.

While Box Constructor shows promise, it still requires several improvements to reach the same level of functionality and usability as tools like Cube Grid.

References

[1] The Level Design Book. https://book.leveldesignbook.com/process/blockout (08.12.2024).

[2] Kramarzewski A., De Nucci E. *Practical Game Design: Learn the Art of Game Design Through Applicable Skills and Cutting-Edge Insights*. Packt Publishing Ltd. 2018.

[3] Totten C.W. *Architectural Approach to Level Design*. CRC Press. 2019.

[4] Mandel T. User/System Interface Design. *Encyclopedia of Information Systems*, 2002, Vol. 1, pp. 1–4.

[5] Finstad K. The usability metric for user experience. *Interacting with Computers*, 2010, Vol. 22, No. 5, pp. 323–327. doi:10.1016/j.intcom.2010.04.004.

[6] Bond J.G. *Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C*. Addison-Wesley Professional. 2014.

Appendix

I — Glossary

CAD Software – Software used to create precise 2D drawings and 3D models using computers.[18]

Constructive Solid Geometry (CSG) – A modeling method used in CAD software that allows users to combine basic shapes, such as cubes and cylinders, using Boolean operations like union, subtraction, and intersection.[19]

Mesh – A collection of vertices, edges, and faces that define the shape of a 3D object in computer graphics.[20]

Plugin – A software extension component that adds new features to an application.[21]

CSGBox3D – A node that allows users to create a box to use with the CSG system.[22]

CSGMesh3D – A node that allows users to use any resource as a CSG shape, provided it is closed and does not self-intersect.[23]

Scene Tree – A class in Godot that manages the hierarchy of nodes in a scene, as well as scenes themselves.[24]

Gizmo – A graphical interface element that allows users to manipulate an object's location, rotation or scale.[25]

Surface normal – A vector that is perpendicular to a surface, representing the direction in which the surface faces.[26]

[18] https://www.3dsourced.com/guides/what-is-cad-guide-modeling/
[19] https://wiki.freecad.org/Constructive_solid_geometry
[20] https://www.geeksforgeeks.org/polygon-mesh-in-computer-graphics/
[21] https://www.geeksforgeeks.org/what-are-plugins/
[22] https://docs.godotengine.org/en/stable/classes/class_csgbox3d.html#description
[23] https://docs.godotengine.org/en/stable/classes/class_csgmesh3d.html
[24] https://docs.godotengine.org/en/stable/classes/class_scenetree.html
[25] https://docs.blender.org/manual/en/latest/editors/3dview/display/gizmo.html
[26] https://mathworld.wolfram.com/NormalVector.html

II — Accompanying Files

The accompanying files are all in a directory "*BoxConstructor-files*". In "*BoxConstructor-files/demo*", the demonstration project that must be imported into Godot 4.4 is located. Opening the Godot project reveals an example scene showcasing a level created with Box Constructor, with the plugin already installed under *"addons/box_constructor*".

In "*BoxConstructor-files/godot*", a standalone Godot 4.4 engine build is provided, which is required to use the plugin.

In "*BoxConstructor-files/plugin-repository*", the complete, up-to-date source code for the Box Constructor plugin is stored as a Git repository. The plugin is also available on GitHub at https://github.com/Hannogert/BoxConstructor and on the official Godot Asset Library at https://godotengine.org/asset-library/asset/3944.

In "*BoxConstructor-files/manual*", the full user manual is provided as a README file. It includes setup instructions, usage examples, and detailed feature descriptions. The same document is also accessible online at https://github.com/Hannogert/BoxConstructor/blob/main/README.md.

In "BoxConstructor-files/questionnaire.pdf", the usability questionnaire used during the evaluation is included as a PDF file, exported from the original Google Form.

In "BoxConstructor-files/user_data.csv", all tester responses are compiled into a single CSV file.

III — License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Hannogert Otti,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis
   Box Constructor – Greyboxing Tool for Godot Game Engine
   supervised by Jaanus Jaggo
2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;
3. am aware of the fact that the author retains the rights specified in points 1 and 2;
4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Hannogert Otti

Tartu, 15.05.2024