Exploring Procedural Level Generation of a Domino Puzzle Game

10th Semester, Medialogy

Project Report

Mikkel Laursen
MTA211042

Aalborg University
Department of Architecture,
Design and Media Technology

Department of Architecture,
Design and Media Technology
Medialogy, 10th Semester
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Exploring Procedural Level Generation of a Puzzle Game based on the Domino Effect

Theme:
Procedural Puzzle Generation

Project Period:
Spring Semester 2021

Project Group:
MTA211042

Participant(s):
Mikkel Laursen

Supervisor(s):
Claus Brøndgaard Madsen

Collaboration:
Victor Skarbye

Copies: 1

Page Numbers: 56

Date of Completion:
May 27, 2021

Abstract:

Procedural content generation for games has segmented itself as an efficient and powerful method of authoring game content. In this project, we attempt to create a development tool for a physics-based path-building puzzle game based on the domino effect. The tool combines procedural generation with the quality of hand-designed game content. The project explores a generation flow that combines state of the art game development tools, procedural puzzle generation methods, and graph theory to create a generation tool that divides the generation process into three stages: Level generation, puzzle generation & puzzle evaluation, at each stage in the generation process the designer can make non-destructive modifications from which the generation process can be continued. Although no evaluation was conducted the generation process of the tool can still be regarded as a success as it reduces development time and the resulting quality significantly

Department of Architecture,
Design and Media Technology
Medialogy, 10th Semester
http://www.aau.dk

AALBORG UNIVERSITET
STUDENTERRAPPORT

Titel:
Udforsking af proceduremæssig bane generation til et puzzle spil baseret på domino effekten

Tema:
Procedural Puzzle Generation

Projektperiode:
Forårs Semester 2021

Projektgruppe:
MTA211042

Deltager(e):
Mikkel Laursen

Vejleder(e):
Claus Brøndgaard Madsen

Samarbejdspartnere:
Victor Skarbye

Oplagstal: 1

Sidetal: 56

Afleveringsdato:
27. maj 2021

Abstract:

Proceduremæssig indholdsgenerering til spil har vist sig som en effektiv metode til at generer spilindhold. I dette projekt forsøger vi at skabe et udviklingsværktøj til et fysikbaseret banebygnings puzzle spil baseret på dominoeffekten. Værktøjet kombinerer proceduregenerering med kvaliteten af hånddesignet spilindhold. Projektet udforsker en genererings process, der kombinerer avancerede spiludviklingsværktøjer, proceduremæssige genereringsmetoder til puzzle spil og graf teori for at skabe et genereringsværktøj, der deler generationsprocessen i de tre faser: Niveaugenerering, puslespilgeneration & puslespilevaluering. I hver fase af generationsprocessen er designeren i stand til at foretage ikke-destruktive ændringer, hvorfra generationsprocessen kan fortsættes. Selvom der ikke blev foretaget nogen evaluering, kan genereringsprocessen af værktøjet stadig betragtes som en succes i det den reducere udviklings tiden og øger kvaliteten af resultatet markant.

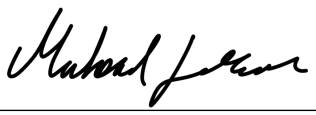
Contents

Preface

This project is a collaboration between Mikkel Laursen (Aalborg University) and Victor Skarbye (Truemax Copenhagen Academy).

We would like to thank our supervisors Claus Brøndgaard Madsen and Paul Ambrosiussen[1] for the great supervision sessions.

Aalborg University, May 27, 2021



Mikkel Laursen
<mlau16@student.aau.dk>

Chapter 1

Introduction

Procedural content generation for games has segmented itself as an efficient and powerful method of authoring game content, both as tools for developers and as fully computer-generated game content found in games such as No Mans Sky[10], Minecraft[14] or Factorio[22] to name some of the more popular titles. In the domain of puzzle games procedural methods have also been utilized, but to a lesser extent. In 2020 Kegel et al. surveyed the relevant work within the puzzle sub-genre, which presents the state of the art procedural content generation methods for the different categories of the puzzle sub-genre.

Procedural content generation is not always a successful way of creating game content as the highly criticized game No Mans Sky experienced after its initial release in 2016, the content of the game was simply too repetitive and did not live up to the expectations of the players. This concern births the question of how procedural generation tools can be utilized effectively while the handcrafted quality of the game does not suffer.

Based on this question this project sets out to create a procedural puzzle generation tool using state of the art and experimental methods to reduce the complexity of creating puzzle levels, while still allowing the designer full control, within the context of a physics path building puzzle game based on the domino effect that is still in its early development phase.

Chapter 2

Background

2.1 Domino The Game

Domino The Game is a Physics Puzzle game that has been in development for approximately 6 months. Its primary mechanic is placing domino bricks with even spacing, that when tipped creates the classic Domino-effect of the bricks tipping the next one in line, and so forth. The idea is to carry the force of the initial push to huge chains of placed Domino bricks.

The Game builds on this mechanic, with special Domino bricks that modify the classic static brick. Currently, there are 5 different modifiers a brick can have, each has its colour to distinguish them, where the brick with no modifier is green

Starter Piece
Colour coded blue, When the player presses "Push" in a level after placing the starter piece, an impulse force will be applied to it, making it fall over thus starting the chain reaction. The direction of this force is visualized by a little arrow, which is visible before pressing the "Push" button. The starter piece can only be placed by the user within the "starter zone".

Pusher Piece
Colour coded yellow, when hit, this brick will have an impulse force applied to it, this makes the brick fall over quickly but more importantly, it will push the next brick in line with a much greater force than usual.

Ball Piece
Colour coded purple, instead of the brick being rectangular, this "brick" is a ball that modifies the natural physical behaviour of the piece, this will allow the brick sequence to transfer the physical chain reaction over greater distances or even

bounce against a wall thus changing direction.

Delay Piece
Colour coded teal, when hit, this block will remain standing for x seconds thus delaying the chain reaction for that long before it applies the physical forces from the prior piece.

Explosion Piece
Colour coded black, when hit, this brick will apply an impulse on every other brick within a radius with a square falloff, thus carrying the chain reaction.

The game presents the player with a level, which is a flat novel maze like street surrounded by walls, within the level, there are domino bricks already scattered around all standing upright. The goal of the game is to tip over all of these pre-placed bricks. A green highlighted rectangle will be found somewhere within the level which is the aforementioned "starter zone". in the case that the level does not already have a starter piece pre-placed within the level, the player will then have to place it within the "starter zone", this is done by tabbing or clicking the button resembling the starter piece found in the Domino toolbox UI seen in Figure 2.1. This action will tell the system that, that tool is selected, then to place the brick, the player has to tab at the desired location and drag the finger to orient the piece to intention, then when lifting the finger or mouse cursor the transformation of the brick will be final.

Figure 2.1: An image of the "Domino Toolbox" from the game, the different tools are colour coded as they are still missing graphics.

Seen in Figure 2.2 is a level used for developing and testing the game, with a simple layout it offers very little in terms of a puzzle. One of the contributing factors of the game that makes it a puzzle type game is the provided pieces in a level. In the case of Figure 2.2, the player has at disposal 30 bricks with no modifiers and one of each modifier type. Given some layout of a level, one could imagine that a combination of these bricks would limit the solution space, however, this constraint alone provides very low dimensional puzzles where the only challenge is to connect the bricks using the least amount of bricks. This is why the game makes

Figure 2.2: An image of placing a starter piece, circled by red. The piece is a shade of transparent green because of the placement being valid, if the piece is overlapping with another piece the domino will turn red. When the cursor or finger is lifted, the domino placement will be final, then turn into the colour code for the type.

use of environmental modifiers, which currently is represented by a pressure plate that can be linked to one or more gates that open up another part of the level, this pressure plate can be activated by constructing a line of bricks, making the last one land on the plate causing it to activate.

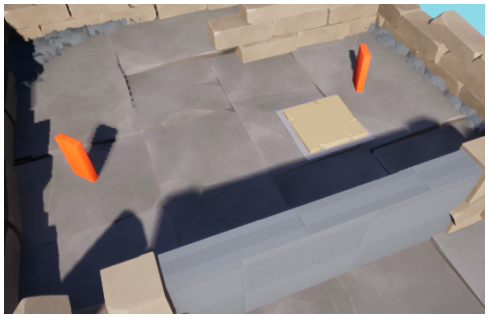


Figure 2.3: Before building a brick line that terminates on the pressure plate.

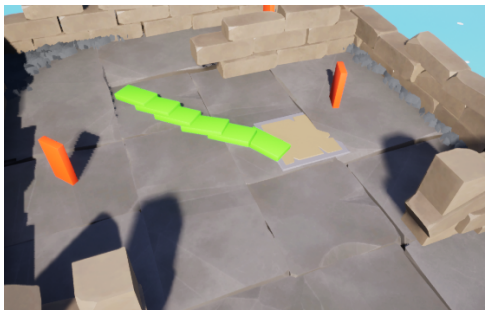


Figure 2.4: Brick line after being pushed over, the gate has been opened.

Figure 2.5: These two pictures visualise building a line that terminates on the pressure plate, the gate opening is an animation where the gate wall moves into the ground which takes approximately 3 seconds to complete.

The addition of the pressure plates and gate, adds the dimension of segmentation

of space. Additionally by having several gates/ pressure plates that have to be opened in a specific order combinatorics can be added to the puzzle space, which in addition to the feature of the fallen over bricks blocking the path where they were placed segments the space even further. The opening animation of the gates as mentioned in Figure **??** takes approximately 3 seconds to complete, along with the transfer of force of the domino effect, adds time as a dimension to the puzzle space.

With the currently presented features of the game, it's possible to make puzzle levels which solutions are at first glance not obvious, where the tighter or more specialized the constraints are in terms of the provided bricks, the fewer solutions to a puzzle level there may exist.
creating levels for such a puzzle game is a time-consuming process that can be divided into four very dependant main concerns:

- Physical boundaries & surface types
- Environmental modifiers layout
- Pre-placed bricks layout
- Provided bricks

Where the physical boundaries & surface types are the area in which the player can place the bricks and the physical simulation is constrained to when the push button has been pressed. The environmental modifiers are the previously mentioned starter zone, the pressure plates and the gates.

2.2 PCG for Puzzle Games

Kegel et al.[5] conducted a survey on the state of the art procedural content generation techniques (PCG) in the domain of puzzle games, to better understand and differentiate puzzle game sub-genres they devised a categorisation which can be seen in Figure 2.6. This categorisation allows one to identify similar puzzle games, and learn from their use of PCG.

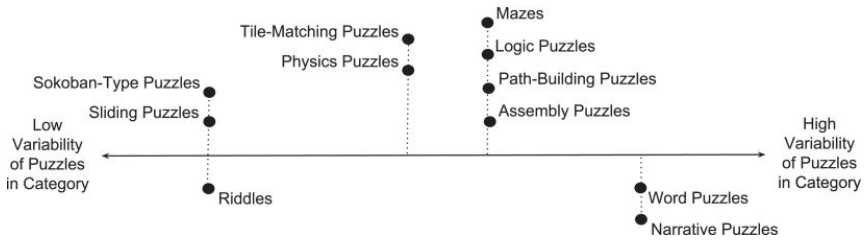


Figure 2.6: Categorisation of puzzle game types from [5], Variability denotes how many puzzles similarly matched left & right differentiates from one another.

Path-Building Puzzle

It's important to note that the authors themselves specify that a puzzle game does not necessarily fit into only one of the categories. In the case of this Domino puzzle game, a combination of a physics puzzle and a path-building puzzle would adequately describe the game. Now, these categorisations are important in the survey, as they are used to summarise the methods and approaches used for procedural generation for the specific categories. With the combination of genres in the Domino game, the important information to extract is the type of generation that was used and the results they achieved. As for the path-building puzzle aspect, Answer set programming (ASP) is very commonly used, an example of the use of ASP is the constrainable automatic level design tools by Smith et. al.[21], where they used ASP to constrain the search space of a generation tool for the game Refraction, the concept of ASP they describe as

> *"Most ASP systems work by translating the programmer-provided problem definition into a low-level, domain-independent representation through the process of grounding (also called instantiation). Then the ground problem is solved by a high-performance combinatorial search algorithm."*[21].

A simplification of ASP is that the method is used on an already existing generation algorithm to constrain and pick from the infinite search space to improve the quality of the PCG content.
One constraint of ASP in terms of this project is that the generation algorithms must exist before ASP can be utilised for its potential.

Another example of a game project that uses ASP in the creation of their levels is a game called Anza Island[4]. Their approach is to create a landscape using common procedural methods, like the Voronoi and Delaunay algorithm. Then they randomly pick a set of specifications that the level should contain, which is fed to an ASP solver called Clingo, the result is a set of specifications for the instantiation of zones and bridges to populate the level.

Physics Puzzle

The Physics part of the Domino puzzle game is an interesting topic which search space is greater than the path-building part of the game, this is also why the state of the art generation algorithms relies on approaches such as evolutionary or genetic algorithms presented in[5] include a paper by Shaker et. al. [19], which implements a clone of the game Cut The Rope, a physics game where the goal is to feed pieces of candy to a frog, this is done by the player triggering elements of the game, firstly by cutting the ropes which the candy is restricted by. Then as the candy is simulating its physics the player can affect it by triggering other game elements like for example rockets. These rockets triggers when clicked, themselves to propel in a direction and asserts forces on objects it hits. In essence, the goal for the player is to find the set of physical interactions at the right timing that makes the candy reach the frog's mouth.

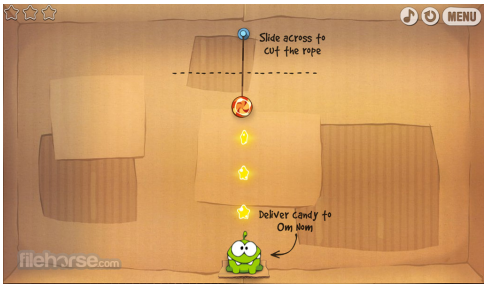


Figure 2.7: Image of a very trivial level in Cut The Rope, swiping the rope with ones finger makes the candy fall into the mouth of the frog.

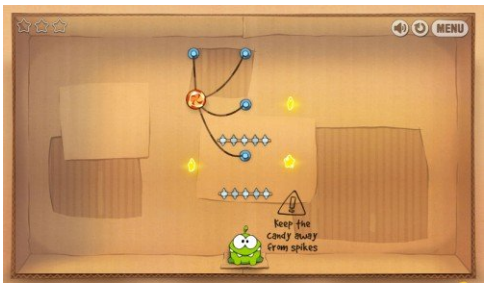


Figure 2.8: Image of a Cut The Rope level, if the candy collides with the yellow stars, the player earns bonus points

Comparing Cut The Rope with the Domino puzzle game, it's apparent that the physical interaction between objects in the Domino context are more uniform and serves the purpose of carrying a tipping force. The Domino bricks with no modifiers could potentially be modelled by algorithms which should guarantee a successful transfer of force, whereas the interactions in Cut The Rope are very dependant on the starting position and rotation of every game element and the timing of the player's interactions.
Following this, the simplification of disregarding the error and randomness that can follow from a solution or layout of domino bricks, without modifiers, that fails

to transfer the force even though the bricks obey the rules of placement offset and rotation could possibly be made.
This simplification could reduce the search space significantly and hints at the eligibility of using algorithmic rules to evaluate and test solutions.

Constructive vs. Generate and test

An important factor to consider when choosing methods of generating puzzles is the distinction between constructive and generate and test methods[5], Constructive algorithms generate and perform validity checks once to create content, whereas generate and test generates content with fewer constraints, then evaluates the generated content and continues to do so until a satisfying solution has been found.
For this project the puzzle generation is not as simple as the examples presented from[5], as the puzzle generation will rely on a level generation algorithm which physical boundaries the puzzle will live within, this dependency complicates the generation methodology, as a change to the level generation has potential to change the puzzle entirely. With this consideration, a constructive approach to the level generation seems as a better approach to make the platform on which to find puzzles using a search based approach like ASP.

Online vs. Offline Generation

When talking about PCG an important distinction to make between the methods is the context in which they run and function. The first type of generation is called online, it refers to the generation that happens after the game has been packaged and built, online methods are commonly found in games with infinite worlds such as Minecraft[14] No Mans Sky[10] or Factorio[22], online methods have to produce very reliable and high-quality results, which in the context of puzzle generation also limits them because of the potential complexity a puzzle can have. Offline methods, on the other hand, can have a higher margin of error as the results can be corrected by a designer before it reaches the user, thus allowing for much more complicated methods. For this project an offline generator seems to be the best fit, as modelling all of the features of the game into a generator in this project may not be feasible and as such hand finishing would be required.
Besides the feasibility, good puzzle design is very much like art, where generated puzzles might be good, the outstanding ones are usually designed or discovered by a designer, as Jonathan blow creator of the game Braid reflects on its development process:

> *"The process of designing the gameplay for this game was more like discovering things that already exist than it was like creating something new and arbitrary."*[2].

As this Domino game is still in development, things are still subject to change. This is why the generation algorithms have to allow the designer to discover the game, differently put the tool we are creating should be an extension of the designer which has the ability to generate a concept or an idea that can be moulded and discovered by the designer.
Practically this means that everything the generator produces should be modifiable by the designer.

Degree and Dimension of Control

As we have established that the generation algorithms should be offline generation and that the output should be modifiable by the designer, it's important to think about the degree and dimensions of control that should go into the algorithms. As we have also established that the algorithms have a larger tolerance of error, the ability for the designer to have a larger variance of output seems sensible. This means that the designer should be able to easily modify the variables that have a clear and direct impact on the generated outcome.

Chapter 3

Concept & Generator Design

In this chapter the concept and conceptual design of the generation tool is presented, the established requirements and design pillars from chapter 2, has influenced many of the tooling and technology choices that were made for this project.

3.1 Concept

As the game is being developed within Unreal Engine 4[9], the generator naturally also has to be developed in the same context to utilize the powerful editor tools that come with the engine. One of the most fundamental tools that come with Unreal is the ability to transform objects via gizmos, which also happens to be an intuitive and control-able way of manipulating objects in 3d environments, these gizmos can be seen in Figure 3.4.

Naturally, this should also be the method of manipulating the generated output of the generation algorithm. This raises the question of how could this work in practice? One solution is to separate the manipulated objects and the produced outcome and use binary space partitioning (BSP) to define the space in which the generated output should exist. BSP is an algorithm used for subdividing space recursively via the use of hyperplanes, the algorithm stores these subdivisions of space in a tree data structure known as a BSP tree. The algorithm of binary space partitioning was developed in 1969 by Schumacker et al.[18], and later extended by Fuchs et al.[7] to virtual 3d objects stored in the BSP tree.

Unreal Engine 4 has an implementation of the BSP algorithm which has a wide variety of uses within the engines systems. One of the systems is known as the Brush system[8], this system was originally designed as a tool for blocking out levels with brushes shown in Figure 3.5. This process is a useful way of ensuring uniform scale between 3D software and the level designer can easily create the

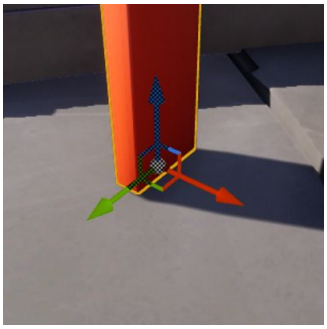


Figure 3.1: Translation gizmo, pressing any of the coloured arrows allows the user to translate the selected object in 3d on the axis of the arrow.

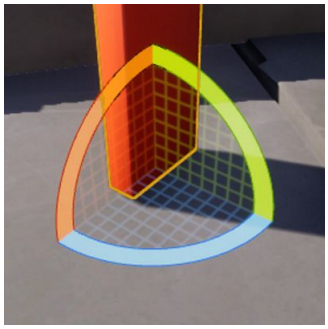


Figure 3.2: Rotation gizmo, pressing any of the colored quarter circles allows the user to rotate the selected object around their axis.

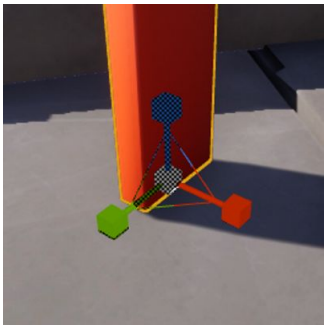


Figure 3.3: Scale gizmo, pressing any of the coloured box handles allows the user to scale the selected object from its pivot point.

Figure 3.4: The basic object manipulation gizmos used in Unreal Engine 4[9].

levels before the final visuals are created.
Another example of the Brush system can be seen in Figure 3.6, where the boolean operations that the BSP tree implement is showcased.

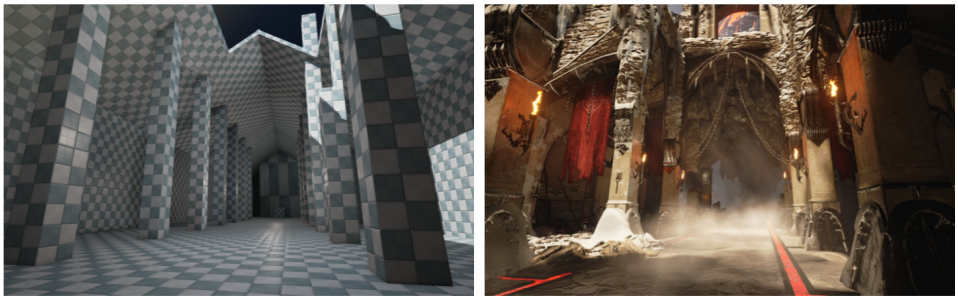


Figure 3.5: An Example of using the Brush system to block out a level seen on the left, the blockout has then been used in a 3D modeling software to create the final 3D models seen on the right. The image is taken from Unreal Engines documentation[8].

This transformation from block-out to the final level is exactly what the puzzle level generator should try to accomplish, where moving a brush in the left image in Figure 3.5, should automatically update the final result seen in the right image in Figure 3.5. Naturally, this presents the next problem, which is how this transformation can be achieved. The problem of generating procedural geometry is very common and has been solved by many games and movies before. Libraries, tools and even entire programs have been created to work with procedural geometry,

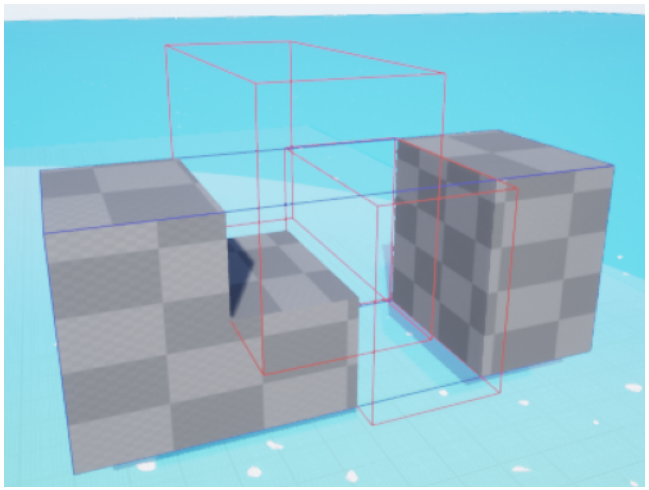


Figure 3.6: Example of the BSP Brush system, red boxes are subtractive and the blue box is additive, these volumes also come in different shapes or compounds, such as cylinders, linear stairs or curved stairs, which are all modifiable by parameters.

one of the industry leaders within this area is a program made by SideFX which was called PRISMS but has since then been renamed Houdini[20]. Houdini was first started back in 1987 and has been developed ever since. Houdini uses a node-based data programming language along with python and a C based language called VEX. The powerful nodes capture the modifiable data that exists within them and allows the user to easily apply pre-coded operations on that data, this allows the realisation of one's ideas very quickly and with powerful visualisation tools for easy debugging.
SideFX has conveniently also made a plugin for UE4 which allows import of the procedural geometry programs named Houdini Digital Assets (HDAs) made within Houdini. These programs can be programmed to accept input in the form of Meshes, Curves and BSP Brushes etc. from UE4 objects to generate their output. These HDA programs will be used in this project for the generated output and the BSP Brushes alongside user-modifiable curves will be generated within UE4 and provided as input for these HDAs. At this point, the discussed generator design can be visualised by the flowchart diagram seen in Figure 3.7.

3.2 Generator Design

With a high-level conceptual understanding of what the generators concerns are, the visual output remains an unanswered question, at least in terms of the aesthet-

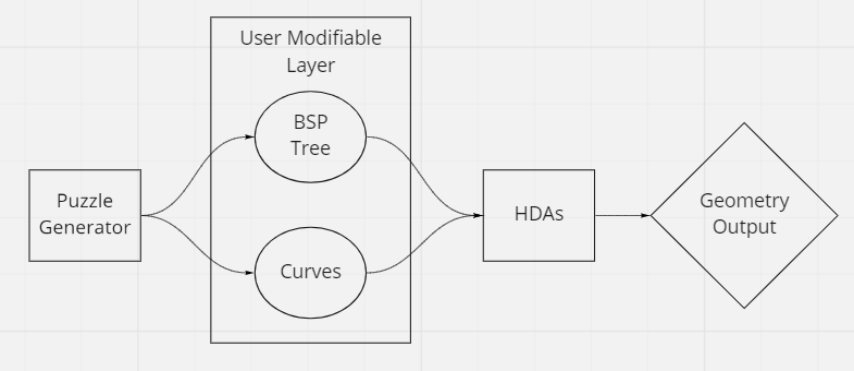


Figure 3.7: The high level flowchart of the level generation, when a user makes a change to the data which feeds into the HDAs, the user can enable automatically recooking the geometry output.

ics and construction. In the early stages of this project, concept art was assembled, which helped us understand how we would like the levels to be constructed and their visual style to be. The first draft of images found on the internet can be seen in Figure 3.8, the final guiding pillars for the visual style which was selected from the draft can be seen in Figure 3.9. To translate this style, it was agreed to create an island to house each of the individual levels, the island should be surrounded with calm water to frame the island, and keep the visual interest focused on the level. The island should consist of a surrounding harbour wall like the top left pictures in Figure 3.9, within the surrounding wall a street layout should mark the area on which the player can place domino bricks, surrounding that street-layout buildings should be placed to create a sense of depth and visual variety, the buildings should also act like a wall which can be used in the physical simulation to aide the player. For the rest of the island, grass, dirt, rock and trees should be scattered to keep the island plane from being too repetitive.

With the graphical elements established, we can divide the HDAs block from Figure 3.7, into sub-generators which has their responsibility. there are three main components which were the island, the streets and the buildings, however since the island and the streets turned out to share the same data, they were merged into one, which then leaves two sub-generators: The house generator and the island generator. To dictate the graphical style of the HDAs geometry output and not couple the final look too tightly to the implemented generator rules we defined a set of modular building blocks which each represents a component of the generated output as inputs for the generators, this way it is very easy to modify the generated output just by changing the input. As for the Island generator, this input was fairly simple, a set of tree meshes, rocks and grass. For the building generator, the list is long, as buildings are made up of floors we wanted some visual variety

Figure 3.8: The draft for the visuals of the levels, these were the images which had some potential from our visual brainstorming session.

between the first, last and middle floors, as such the user can specify for each of the floors: windows, walls and pillars which has to adhere to 2x2 meter segments. For the first floor, the user can specify doors and a porticus additionally. For the roof, a setting controls whether the roof is tiled or flat with a fence, here the user can specify the tile, fence post and fence railing. A setting also controls whether the windows should have overhangs and or flower boxes. Finally, a list of props that will be scattered around the building can be specified, for this project we made a wooden barrel and a wooden cart. The final visual result excluding the streets can be seen in Figure 3.10, it's important to note that the generated geometry, in this case, is based on hand-placed BSP brushes and is independent of the puzzle generator.

Referring back to chapter 2, four dependant concerns of the generation was presented, we see that apart from the physical boundaries and surface types solved by the HDAs, three remain to be delegated. As for the puzzle generation, the Environmental modifier's layout and pre-placed bricks layout is concerned with the puzzle

Figure 3.9: The final visual inspiration for the game. The upper two pictures to the left capture the setting and simplicity that we want to strive for, the lower-left picture has some fantasy elements in the architecture that we want to try and incorporate with the somewhat roman style of the middle picture. Rightmost is the colour palette that we decided on inspired heavily from Studio Ghibli[11].

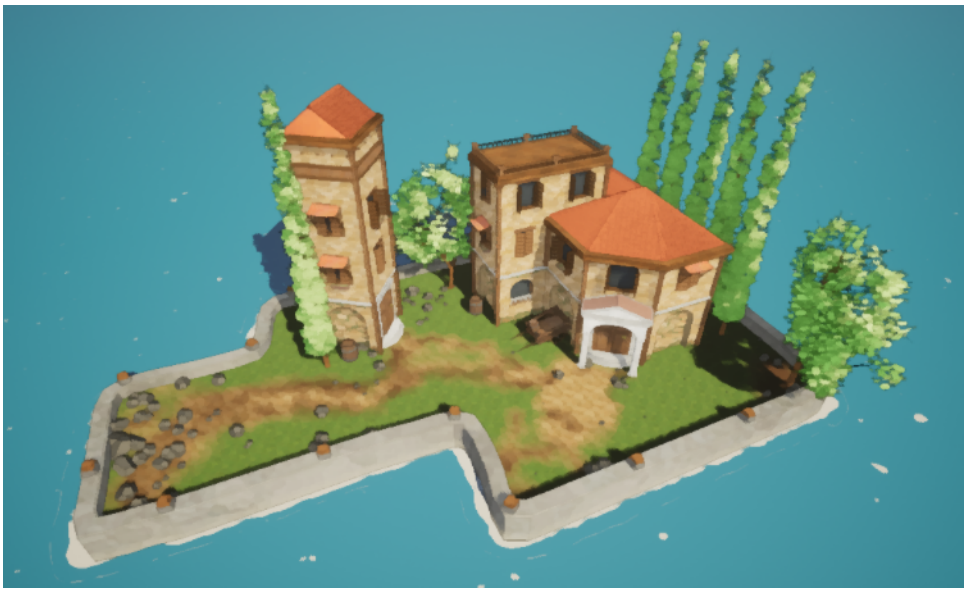


Figure 3.10: A showcase of the visuals of the game, hand-placed BSP brushes are manually specified for the HDAs, in this image the streets is not added and the trees are also hand-placed.

generation, one issue that has not been discussed yet is a small amount of error that is introduced in the island HDA, which alters the output geometry slightly from the inserted BSP brushes, this transformation requires the puzzle generator block from Figure 3.7, to be split into a level generation block and a puzzle generation block which should be executed after the outputted geometry. The last concern which was the provided bricks is very important for a meaningful level, the constraints of the puzzle are what makes it meaningful, this is also why an entire block concerned with verifying the validity and evaluating the level should be executed

after the puzzle generation. Updating Figure 3.7, with the additional consideration of the two HDAs and the evaluation step we can update the flowchart as seen in Figure 3.11.

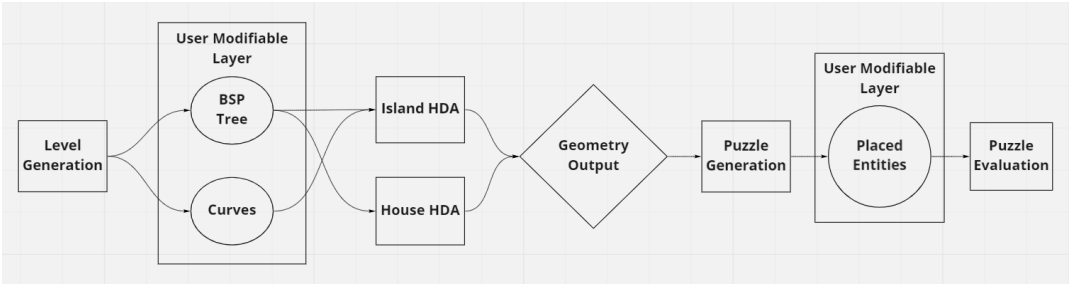


Figure 3.11: Final flowchart of the generation process and its components.

Chapter 4

Implementation

In this chapter we will present how the generators were implemented, besides the used algorithms, the generator uses a combination of technology in which context the presented generator flow from Figure 3.11, has to be implemented. Using and modifying existing software can present interesting challenges and this project was no different.

4.1 Bridging Houdini & Unreal

As discussed earlier in chapter 3, the generation was delegated between Unreal Engine 4 and Houdini Engine, this meant that for it to work in the editor context of Unreal engine we needed to use Houdini Engines plugin for Unreal Engine, now the developers at SideFX whom created this plugin has created a nice way for a user to specify parameters, input meshes and level data, such as the curves and BSP Brushes, however, the interface which supplies this data to Houdini Engine is only exposed privately and when the link between the two programs is established. This meant that for another part of the program to supply this data we had to extend the plugin. Luckily the source code is freely available to read and modify if one so desires within Unreal Engine, unlike other big modern game engines. Now, the functionality we need is to supply inputs via. what Houdini Engine calls "world selection" and to specify the mesh inputs to the HDAs, on top of that we also need to be able to respond to some messages to carry and time the program flow, the first one is when a Houdini asset component is done baking and the last one is when the Houdini asset component has been supplied new parameters. To add this functionality we can add the function and event definitions to the file located within the plugins folder in HoudiniEngineRuntime, called HoudiniAssetActor.h seen in listing 1.

Now, the implementation of the functions from listing 1 are pretty trivial, for Up-

```
UFUNCTION(BlueprintCallable, Category="Houdini Asset Component")
bool UpdateWorldSelectionInputs(FString Name, const TArray<AActor*>& Actors, int Index=0);

UFUNCTION(BlueprintCallable, Category="Houdini Asset Component")
void SetMeshInputs(TArray<FNamedMesh> Meshes);

UFUNCTION(BlueprintImplementableEvent)
void OnHoudiniAssetComponentPostCookBake();

UFUNCTION(BlueprintNativeEvent)
void OnHoudiniAssetComponentPreCookSupplyParams();
```

Listing 1: Function and event declarations needed for the UE4 generator to communicate with the HDAs. UFUNCTION() is a macro that tells UE4s c++ based reflection system to expose the function in the specified context, in this case "BlueprintCallable" or "BlueprintImplementableEvent" to UE4s visual scripting language called Blueprint, which is used to assemble the generators execution flow from a high level architecture.

dateWorldSelectionInputs, it sets the type of input to EHoudiniInputType::World and the HoudiniAssetComponents input is set to the TArray of Actor references. For the SetMeshInputs function, it takes a structure with a FName (string type) that should match the mesh inputs name you want to set, and the actual asset reference it should be set to. The last two are delegation functions, meaning that they can be overridden via. the reflection system in Blueprint, the difference between BlueprintNativeEvent and BlueprintImplementableEvent, is that the native event can also have an implementation in the c++ layer.

4.2 High Level Architecture

To implement the functionality from the flowchart in Figure 3.11, the best way to achieve this is to utilise blueprints, as they are not constrained by Unreal Engines module structure, the module structure in unreal is a way of reinforcing good design patterns in terms of the generated dependency graph, however, this module structure also limits any dependency across modules. A plugin lives within its module, a default module is provided for the game code. This means that for these modules to call into each other, blueprints is the intended way to do so. The type of blueprint that is the best fitting for our purposes is called a Bluetillity, a Bluetillity is an editor UI widget blueprint that can be used to create editor windows and contain code for the buttons, sliders etc. that they contain. The Bluetillity user interface that is used in this project can be seen in Figure 4.1.

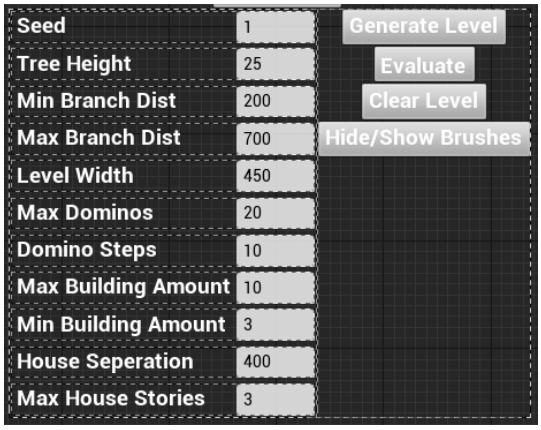


Figure 4.1: The editor UI created for the Bluetillity, used by the designer to call into the generators.

Apart from the parameters, the designer can tweak to modify the output, the far most important element in the editor user interface is the "Generate Level" button, which if we look at the OnClick Event implementation seen in Figure 4.2 it demonstrates the architecture earlier described where the Blueprint layer calls into the underlying exposed c++ code.

The call order is not as simply implemented as it is shown in Figure 3.11, rather it

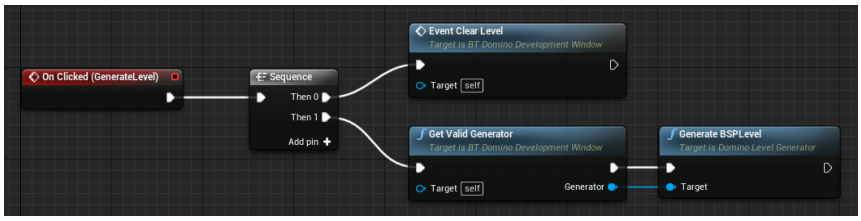


Figure 4.2: The Generate level Blueprint event implementation, we see that the event first calls an event called clear level, then calls an event to get a valid generator and calls the GenerateBSPLevel function on the returned object, which is the exposed UFUNCTION from the DominoLevelGenerator C++ class, see listing 2.

```
UFUNCTION(BlueprintCallable)
void GenerateBSPLevel();
```

Listing 2: The UFUNCTION GenerateBSPLevel has been exposed by the UE4 c++ reflection system and is called by the Blueprint graph seen in Figure 4.2.

is a call-stack which starts from the GenerateBSPLevel Event, the final functionality of that c++ function spawns the two HDAs, now the HDAs are Blueprint actors that extent from the earlier mentioned HoudiniAssetActor class. This means that we can extend or override the functionality of the HDA object, which we want. The

functions we declared in 1, can now be used to provide the desired data and input to the HDA by the event OnHoudiniAssetComponentPreCookSupplyParams, which is called when the default parameters or input is supplied.
The creation of the HDAs takes some initialisation time and the next step in the generation flow requires the generated geometry, here the earlier mentioned event OnHoudiniAssetComponentPostCookBake, allows us to continue the generation process, the Blueprint event can be seen in Figure 4.3. Now the function that is called deviates from the final flowchart, an additional step of baking a Navigation mesh that UE4 uses for their AI systems, is used by some of the implemented algorithms, their specific uses shall become apparent later. However, the process of baking the navigation mesh is implemented as an asynchronous routine and as such only after the bake has been completed is the Puzzle Generation function called.

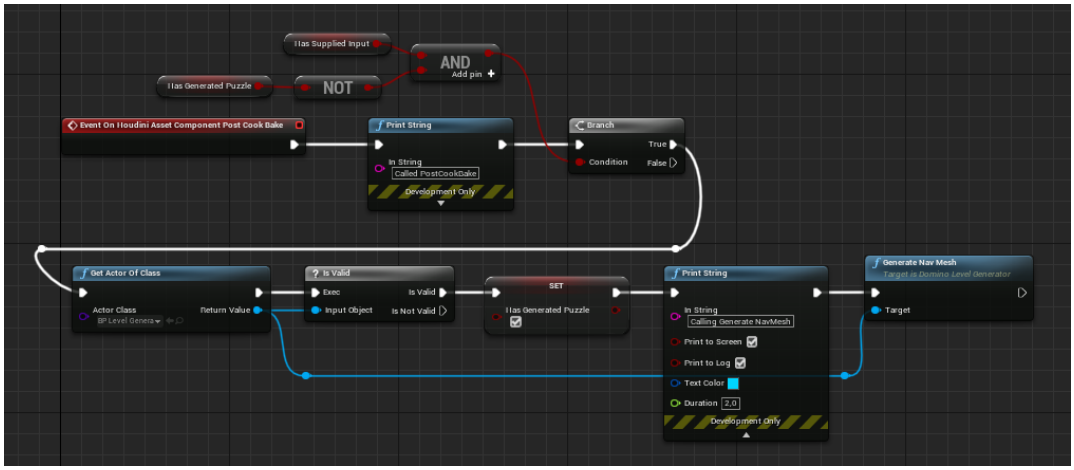


Figure 4.3: Blueprint code that is executed when the OnHoudiniAssetComponentPostCookBake event is fired, the code is from the Blueprint extension of the Island HDA.

4.3 Level Tree Generation

The first algorithm of the level generation process is responsible for creating the data used to place the first user-modifiable layer. The algorithm works like a simple tree, where at each node, there's a configurable probability of N branches. A height of the tree is specified before the generation, until that height is reached the tree will continue to grow. There are also some simple rules governing the growth of branches, the branches can only grow in a random direction constrained by a cone, which centre is parallel to the difference vector of the parent node

and the leaf node on which we are standing. The angle of this cone is given by $\frac{\pi}{2}$. The length of the branch is defined by a random unit scalar which is then remapped to a user-controlled range, which parameters are called *MinBranchDist* and *MaxBranchDist*. The pseudo-code for the algorithm can be found in Figure 4.9. This tree generation algorithm is used in the function called GenerateBSPLevel, as described in Figure 4.2. The tree is then used to guide the placement of the BSP Brushes, which is placed on every edge and every vertex of the Level Tree, the algorithm used to spawn the BSP Brushes can is seen inFigure 4.10. Example results of the two described algorithms can be seen in Figure 4.8. The BSP Brushes are placed on all of the edges, at sharp corners this leaves a wedge gap, to fill this gap a BSP Brush is also placed on every vertex.

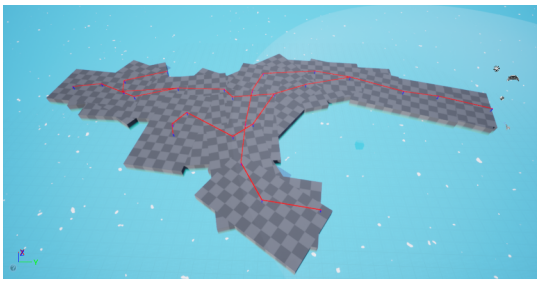


Figure 4.4

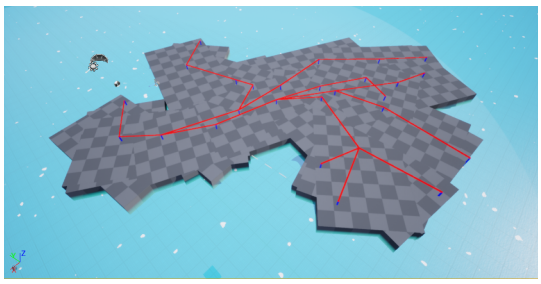


Figure 4.5

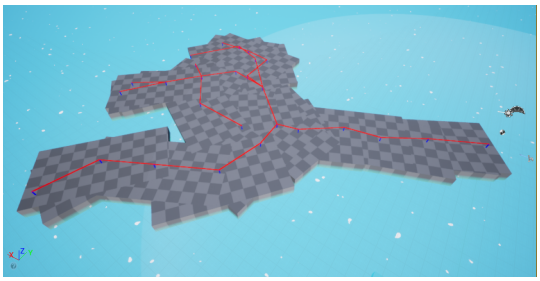


Figure 4.6

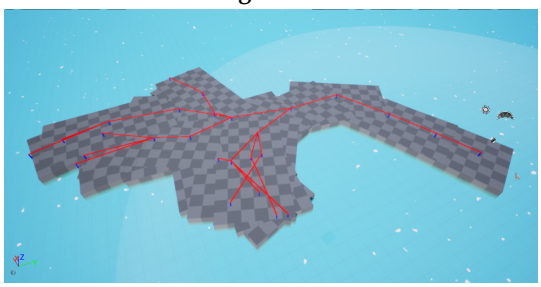


Figure 4.7

Figure 4.8: Examples of the Generate BSP Level algorithm. The red lines are the generated tree edges from the Generate level tree algorithm seen in Figure 4.9. Note that the intersections of the edges has deliberately been accepted, as limiting the generation with too many rules will limit the amount of random outcomes.

Algorithm 1: Generate Level Tree

```
input: A Pointer to a container object of the LevelTree
Compute first branch of the tree;
RandScalar ← RandomInRange(MinBranchDist, MaxBranchDist);
RandVector ← RandUnitVector();
RootEnd ← RandVector ∗ RandScalar;
CurrentLeaves ←(empty list);
Start ← TreeVertex(zeroVector, zeroVector);
End ← TreeVertex(RootEnd, (RootEnd − zeroVector).Normalize());
End.Parent ← Start;
Start.Children.Add (End);
LevelTree.Vertices.Add(Start);
LevelTree.Vertices.Add(End);
LevelTree.Edges.Add(TreeEdge(Start, End));
CurrentLeaves.Add(End);
CurrentHeight ← 0;
Compute all other branches of the tree;
while CurrentLeaves.Num() > 0 do
    NewLeaves ←(empty list);
    for Leaf in CurrentLeaves do
        Rand ← RandomInRange(0, 1.0);
        Compute number of branches at node;
        N ←WeightedRandomFromBranchWeights();
        for i ← 0 to N do
            if CurrentHeight >= TreeHeight then
                return;
            end
            PointInCone ← RandPointInCone(Leaf.Direction, π ∗ 0.5);
            RandConeScalar ←
             RandomInRange(MinBranchDist, MaxBranchDist);
            PointInCone ← (PointInCone ∗ RandConeScalar ) + Leaf.Position;
            NewVertex ← TreeVertex(PointInCone, (PointInCone −
             Leaf.Position));
            NewVertex.Parent ← Leaf;
            LevelTree.Vertices.Add(NewVertex);
            Leaf.Children.Add(NewVertex);
            LevelTree.Edges.Add(TreeEdge(Leaf, NewVertex));
            NewLeaves.Add(NewVertex);
            CurrentHeight ← CurrentHeight +1;
        end
    end
    CurrentLeaves ← NewLeaves;
end
```

Figure 4.9: Generate Level Tree Algorithm

Algorithm 2: Generation of the BSP Brush Level

```
Tree ← LevelTree();
GenerateLevelTree(&Tree);
Brushes ← (empty list);
for Edge in Tree.Edges do
    ABDif ← Edge.GetA().Position − Edge.GetB().Position;
    ABAvg ←(Edge.GetA().Position + Edge.GetB().Position) ∗ 0.5;
    DimX ← ABDif.Size();
    Forward ← ABDif.Size();
    Rot ← MakeRotFromXZ(Forward, UpVector);
    Brush ← SpawnBSPBrush(BrushType :: Add, Vector(DimX, LevelWidth,
     100));
    Brush.SetActorLocation(ABAvg);
    Brush.SetActorRotation(Rot);
    Brushes.Add(Brush);
end
for Vertex in Tree.Vertices do
    if Vertex == nullptr or Vertex.Parent == nullptr then
        continue;
    end
    if Vertex.Children.Num() >= 1 then
        for Child in Vertex.Children do
            VertexChildDir ← (Child.Position − Vertex.Position).Normalize();
            VertexParentDir ← (Vertex.Parent.Postion −
             Vertex.Position).Normalize();
            AvgDir ← (VertexParentDir + VertexChildDir) ∗ 0.5;
            Rot ← MakeRotFromXZ(AvgDir, UpVector);
            OrthoDir ← VertexChildDir ×UpVector;
            OrthoDirScaled ← OrthoDir ∗ LevelWidth;
            ProjOnAvgDir ← OrthoDirScaled.ProjectOnTo(AvgDir);
            DimX ← ProjOnAvgDir.Size() ∗ 2;
            Brush ← SpawnBSPBrush(BrushType :: Add, Vector(DimX,
             LevelWidth, 100));
            Brush.SetActorLocation(Vertex.Position);
            Brush.SetActorRotation(Rot);
            Brushes.Add(Brush);
        end
    end
end
```

Figure 4.10: Generation of the BSP Brushes that define the level

4.4 House Generation

The next concern of the level generation is the houses, which like the Island generation is based on BSP Brushes. The placement of these Brushes has to conform to the existing level layout, with a limited margin of error in terms of the degree to which the house overlaps with the BSP Brushes. An algorithm is therefore required to find the placements such that the error is minimised. Once the placement is found the next algorithm should divide the selected space into a $2x2$ meter grid which is randomly populated with Brushes based on the desired volume percentage of the total volume of the space. These Brushes will be used to generate the houses. Brushes will also be generated for the foundation of the houses which will later be used to make the overlapping area of the island conform to the houses, such that the street layouts curb described in chapter 3, will flow around the houses.
The first algorithm used for placing the houses bases the placement on the level tree, where each node of the tree acts as a socket for a house to be placed on, thus occupying that socket or node index. The pseudo-code for the house node indices generation can be seen in Figure 4.11.

Algorithm 3: Generate House Indicies

Output: A List of Integers corresponding to random indices in the LevelTree

```
RandomIndices ← (empty list);
VertCount ← Tree.Verticies.Num() − 1;
if AmountHouses > VertCount then
    return RandomIndices;
end
Indices ← (empty list);
for i ← 0 to VertCount do
    Indices.Add(i);
end
while RandomIndices.Num() != AmountHouses do
    Sample ← RandomInRange(0, Indices.Num() − 1);
    RandomIndices.Add(Sample);
    Indices.RemoveAt(Sample);
end
return RandomIndices;
```

Figure 4.11: Algorithm that randomly picks indices in the LevelTree to be used as sockets for houses to be generated.

The second part of the house generation algorithm is a continuation of the GenerateBSPLevel function shown in Figure 4.10, this algorithm takes the random indices, from that it then generates a point in space on which we define a volume in which the house should be generated. This is done by picking a random point within a torus, then evaluating the distance to all of its neighbouring nodes(neighbouring nodes are in this case defined by Euclidean distance) and centres of edges, if all of those distances lie between a user-defined lower and upper bound denoted *HouseMinDistance* and *HouseMaxDistance*, the point is then added as added valid point and can be used to generate a house. The house point and extent generation algorithm is shown in Figure 4.16. Generating the actual Brushes that make up the house is a process that involves some math to place the boxes on the $2x2$ grid and random number generation which will be randomly limited to constrain the amount of volume a sub-box can occupy. The resulting generated level so far can be seen in Figure 4.15.

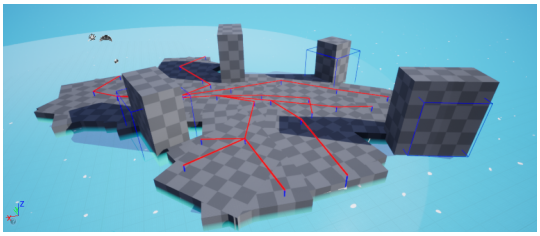


Figure 4.12

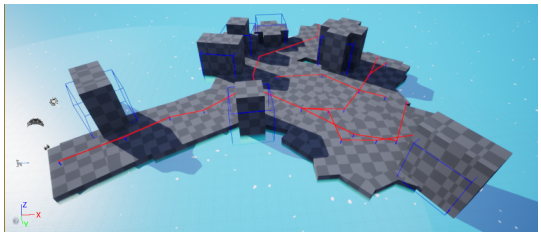


Figure 4.13

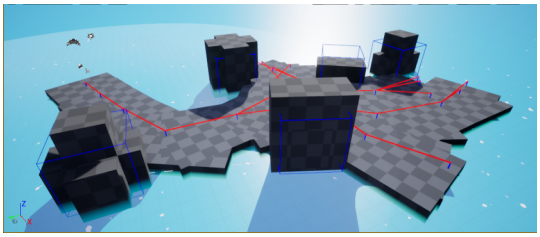


Figure 4.14

Figure 4.15: Examples of the BSP Level Generation with houses, the houses are neatly place along the perimeters of the level, only overlapping slightly.

Algorithm 4: Generate House Locations and extents

```
HousePoints ← (empty list);
for I in Indices do
    Vertex ← Tree.Vertices[I];
    Get the nodes, and edges within 2000cm of the Vertex;
    Neighbours ← GetAllSubdividedNeighboursByDistance(Vertex, 2000);
    Point ← RandomUnitVector();
    Point ← Point * RandomInRange(HouseMinDistance, HouseMaxDistance);
    Valid ← IsBulidingPointValid(Vertex.Position + Point, Neighbours,
     HousePoints, HouseMinDistance);
    Tries ← 50;
    Itr ← 0;
    while ! Valid do
        Point ← RandomUnitVector();
        Point ← Point * RandomInRange(HouseMinDistance,
         HouseMaxDistance);
        Valid ← IsBulidingPointValid(Vertex.Position + Point, Neighbours,
         HousePoints, HouseMinDistance);
        if Itr > Tries then
            Success ← false;
            break;
        end
        Itr ++;
    end
    if ! Success then
        continue;
    end
    HousePoints.Add(Vertex.Position + Point);
    Child ← Vertex.GetRandomChild();
    AvgLength ← 0;
    if Vertex.Parent != nullptr and Child != nullptr then
        AvgLength ← Distance(Vertex.Position, Vertex.Parent.Position) +
         Distance(Vertex.Position, Child.Position) * 0.5;
    else if Child != nullptr then
        AvgLength ← Distance(Vertex.Position, Child.Position);
    else if Vertex.Parent != nullptr then
        AvgLength ← Distance(Vertex.Position, Vertex.Parent.Position);
    end
    Brush ← PlatformBrushes [I];
    XDim ← AvgLength * 0.5 * RandomInRange(0.75, 1.0);
    YDim ← LevelWidth * 0.5 * RandomInRange(0.75, 1.0);
    ZDim ← (100 * RandomInRange(1, MaxHouseStories)) + 50;
    Position ← Vector(Vertex.Position.X + Point.X, Vertex.Position.Y +
     Point.Y, Vertex.Position.Z + Brush.Z + ZDim − 100);
end
```

Figure 4.16: This algorithm generates a point adjacent to the LevelTree, the point is only used to generate a house if its distance to all neighbouring nodes and edges is within a lower and upper bound. From this point, we then calculate the extents XDim, YDim, ZDim which we want to generate a random house within.

4.4.1 House HDA

The HDAs was implemented by Victor the collaborator of this project with consultancy from the author, and as such the credit shall go to him. The House HDA is a complicated interdependent generation program that we will not explain on a deep technical level in this report. Rather a flowchart showcasing the rough process using simple explanations of the methodology can be seen in Figure 4.17.

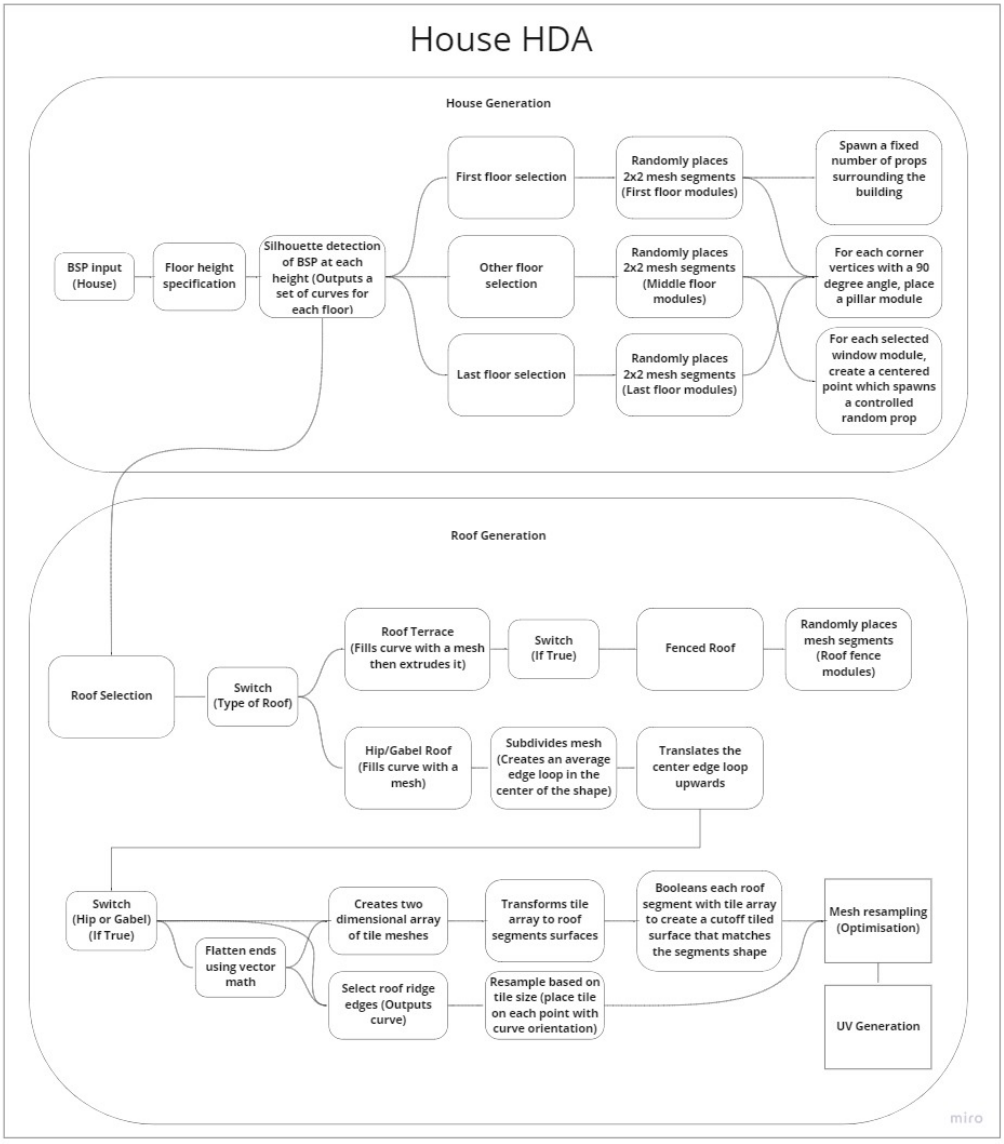


Figure 4.17: Flowchart crudely explaining the methodology behind the House HDA.

4.5 Island Generation

With the BSP Level fully generated we can begin to tackle the problem of generating an island that encompasses the level, the way we decided to tackle this problem was to generate a curve with modifiable control points, such that the result can still be edited by the designer. The problem of generating a bounding curve from a set of vertices and building points can be solved by computing an ordered map of dot products between a reference vector that goes from the bounding box centre of all the points to a chosen reference point by rotating the reference vector incrementally around the centre we can divide a circle into bins, these bin's points can be found by comparing the dot product of all other vectors from the bounding centre to the reference vector, if they lie within the percentile of divided points given by: $\frac{1}{NumBins} * 0.5$ meaning that the dot product can not be less than $1 - (\frac{1}{NumBins} * 0.5)$. The radial dot sorting algorithm can be seen in Figure 4.19. Then using these points, we can create a curve by selecting the furthest point from the centre of each bin. The curve creation algorithm can be seen in Figure 4.20. An example of a generated curve can be seen in Figure 4.18.

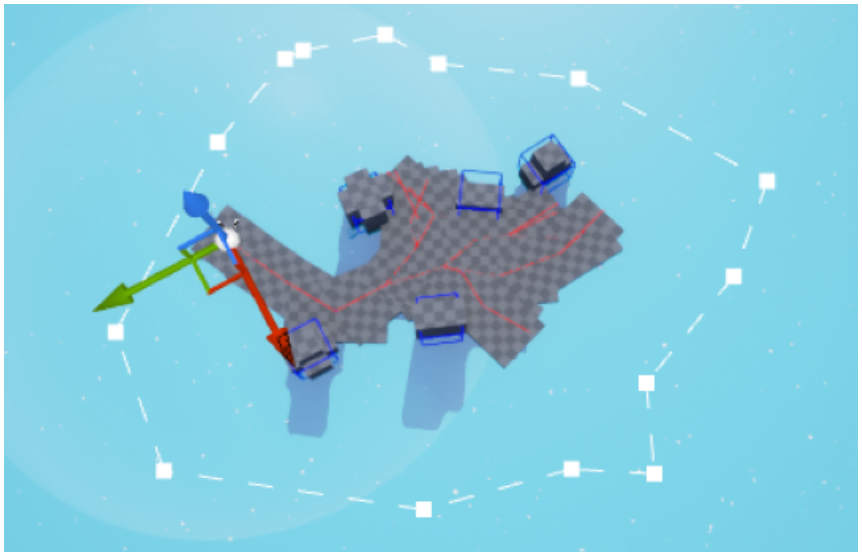


Figure 4.18: This image shows a top-down view of the generated curve around a level.

Algorithm 5: Radial Dot Sorted Points

```
input : A Centre Vector, amount of bins
output: A Map of bins and their set of points
Map ← (empty map);
DotStepSize ← (1.0 / BinDivisionCount) * 0.5;
RotationStepSize ← 360 / BinDivisionCount;
RotatedDotVector ← (Center − Tree.Vertices[0].Position).Normalize();
for i ← 0 to BinDivisionCount do
    if i != 0 then
        RotatedDotVector ← RotateAngleAxis(RotationStepSize, UpVector);
    else
        RotatedDotVector ← RotateAngleAxis(RotationStepSize * 0.5,
         UpVector);
    end
    Pts ← (empty list);
    for Vert in Tree.Vertices do
        Dir ← (Center − Vert.Position).Normalize();
        if Dir × RotatedDotVector > 1 − DotStepSize then
            Pts.Add(Vert.Position);
        end
    end
    for HousePt in HouseLocations do
        Dir ← (Center − HousePt).Normalize();
        if Dir × RotatedDotVector > 1 − DotStepSize then
            Pts.Add(HousePt);
        end
    end
    Map.Add(i, Pts);
end
return Map;
```

Figure 4.19: This algorithm sorts a cluster of points by finding the bounding box centre, then by rotating a vector around a circle based on the centre we divide the circle into bins where the points with very similar dot products to the rotated vector will be added.

Algorithm 6: Curve Creation

```
Pts ← (empty list);
for i ← 0 to Map.Num() do
    SmallestDist ← 0;
    FurthestPoint ← ZeroVector;
    if Map.Vertices.Num() > 0 then
        for Pt in Map.Vertices do
            if NewDist > SmallestDist then
                SmallestDist ← NewDist;
                FurthestPoint ← Pt;
            end
        end
        if FurthestPoint != ZeroVector then
            Map.Add(FurthestPoint);
        end
    end
end
ScaledPts ← ScaleVectorArray(Pts, Center, IslandScale);
Sets the UE4 Spline Class points;
SetSplinePoints(ScaledPts);
```

Figure 4.20: This algorithm creates the curve that is used to specify the extents of the island walls in the Island HDA.

4.5.1 Island HDA

The HDAs was implemented by Victor the collaborator of this project with consultancy from the author, and as such the credit shall go to him. The Island HDA will likewise not be explained on a deep technical level in this report. Rather a flowchart showcasing the rough process using simple explanations of the methodology can be seen in Figure 4.21.

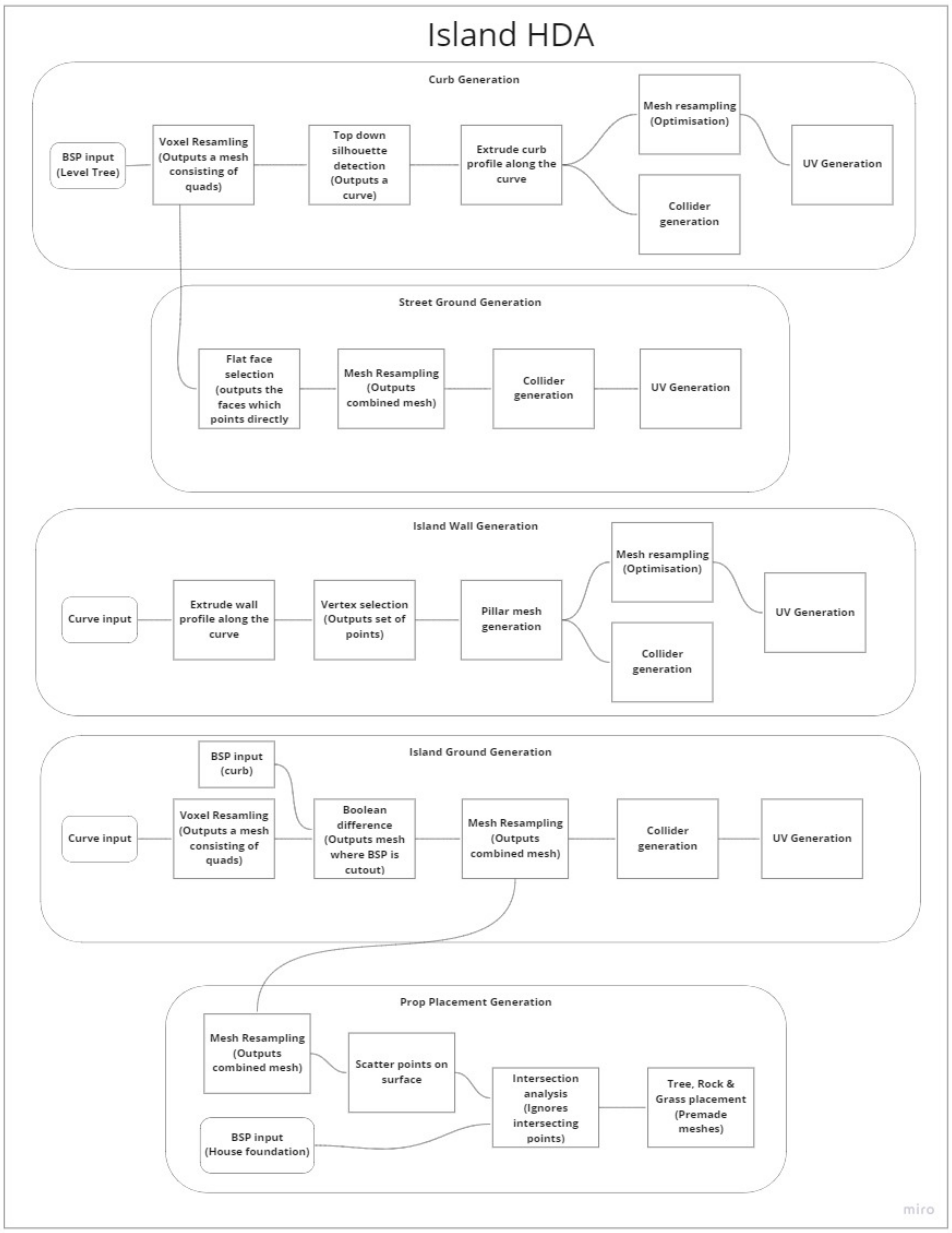


Figure 4.21: Flowchart crudely explaining the methodology behind the Island HDA.

4.6 Puzzle Generation

As the Island and House HDAs has completed their generation of the geometry, it is time for the generation of a puzzle that is more clearly visualised in Figure 3.11 on the level. Now this puzzle generation has two concerns which are the second

and third item in the list of concerns see chapter 2, namely the environmental modifiers layout and the layout of the pre-placed bricks. In the projects current state, the generation of these two concerns is trivial in terms of the complexity of the generated puzzle. Currently, the pre-placed bricks has no modifiers and the only environmental modifier included in the gates and pressure plates seen in Figure **??**. However, the process of creating a layout where the elements are placed to some degree of an informed decision quickly becomes non-trivial. One causing factor of the complexity is the unconstrained growth of the level tree, where intersecting edges are allowed. The intersecting edges can cause branches to grow together creating connections in the level, which as far as the tree is concerned are not there. With the current data representation of the level, it is therefore hard to compute subdivisions within the level as would be required for a gate and the pressure plate to fulfil their purpose. Or to compute an even distribution of the pre-placed domino bricks.

The current solution involves computing an undirected connectivity graph where it is possible to travel in more or less a straight line from a given node to its neighbours on the generated geometry. Now the travelling from one node to its neighbour is a problem that requires knowledge about the generated mesh, luckily UE4 provides a solution to this problem as also described earlier in this chapter. It is possible to generate a navigation mesh, which can be configured to generate on specified geometry and to follow specified rules. With this navigation mesh, UE4 has implemented path-finding algorithms that can be used to find paths from a location on the mesh to another. The path consists of a list of corner points required to travel on the navigation mesh to the destination. If it's possible to traverse from point A to point B in a straight line the resulting list of corner points will only contain the point A and B.
In addition to computing the connectivity graph, we also want to compute whether an edge is part of a cycle, ignoring the cycles between connected nodes given the graph is undirected. The acylic edges will be used later as the connecting edge on which we want to place the gates.

4.6.1 Connectivity Graph Algorithm

With the navigation mesh, we can compute the earlier mentioned connectivity graph by computing the paths from a node to all other nodes, if the path obeys the following rules, then we can accept them as connected.
If the path from node A to node B has a total length that is at a maximum 5% greater than the maximum branch distance, the total length is less than or equal to the euclidean distance and if the path from node A to node B consists of less than 5 corner points we can accept them as connected. The connectivity graph creation

algorithm can be seen in Figure 4.22, and the result of the algorithm can be seen in Figure 4.23.

Algorithm 7: Connectivity Graph Creation Algorithm

input : The Level Tree
output: A Graph from the Level Tree, where the edges are recomputed by connectivity.

```
NavigationSys ← GetCurrentNavigationSystem();
Vertices ← (empty list);
Edges ← (empty list);
for i ← 0 to Tree.Vertices.Num() do
    Vertices.Add(CreateVertex(Tree.Vertices[i].Position,
     Tree.Vertices[i].Direction, i));
end
for i ← 0 to Vertices.Num() do
    for j ← 0 to Vertices.Num() do
        if i == j then
            continue;
        end
        EuclidianDistance ← Distance(Vertices [i].GetLocation(), Vertices
         [j].GetLocation());
        NavigationPath ← NavigationSys.FindPathToLocation(Vertices
         [i].GetLocation(), Vertices [j].GetLocation());
        Length ← NavigationPath.GetPath.GetLength();
        if Length == 0.0 then
            continue;
        end
        if Length > MaxBranchDist * 1.05 then
            continue;
        end
        if Length <= EuclidianDistance and
         NavigationPath.GetPathPoints.Num() < 5 then
            Edge ← CreateEdge(Vertices [i], Vertices [j], Length);
            Edges.Add(Edge);
            Vertices [i].Edges.Add(Edge);
        end
    end
end
return ← Graph(Vertices, Edges);
```

Figure 4.22: This algorithm takes the Level Tree as input, it then computes the connectivity of the vertices in the tree by using UE4's navigation systems pathfinding algorithm on the generated navigation mesh, to find the path length, it then compares this path-length to the euclidean distance between the vertices. If the path length is less than the euclidean distance and the path consists of less than 5 points then we create an edge between the two vertices. Finally, a Graph of the new vertices and edges is returned.

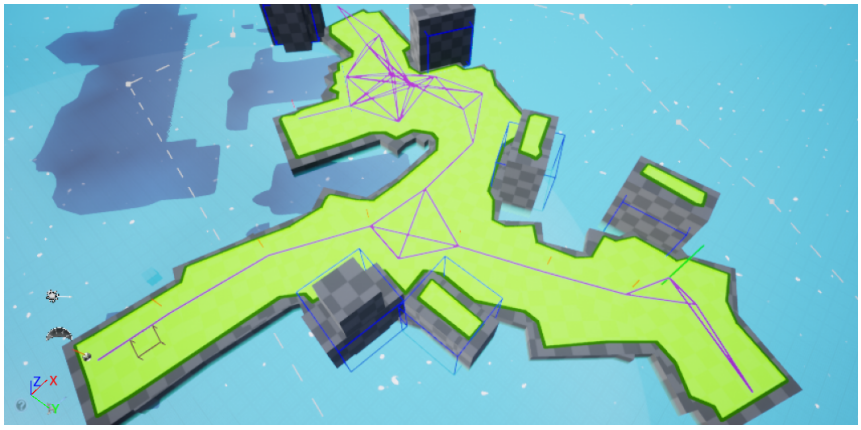


Figure 4.23: This image shows the connectivity graph (purple lines) that was computed from the Level Tree, the green surface is the generated navigation mesh that was used to determine whether the two nodes are connected.

4.6.2 ACylic Edge Detection

The problem of detecting if a graph has a cycle is commonly solved by either using a DFS(depth-first search) or BFS(breadth-first search), in this case, we know from the nature of the connectivity algorithm and visual proof see Figure 4.23 that many of the edges will be cyclic edges, armed with this knowledge, the optimal approach for this application is to use the BFS approach, as many of the cycles will be found in the proximity of the concerning edge. To detect the cyclic edges we start at an edge, we then add one of the vertices to a queue, by traversing all edges connected to that vertex and so on, only visiting a vertex once, we try to reach the other vertex of the original edge. If we can reach the other node, the edge is cyclic if not the edge is acyclic. The implementation of the algorithm can be seen in Figure 4.24.

Algorithm 8: Acyclic Edge Detection

```
input : The Edge to test if cyclic
output: true or false
VFlags ← (empty list);
for i ← 0 to TotalVertCount do
    VFlags.Add(false);
end
VFlags [Edge.GetA().ID] ← true;
VertQueue ← (empty queue);
VertQueue.Enqueue(Edge.GetA());
while ! VertQueue.IsEmpty() do
    Vert ← VertQueue.Peek();
    for E in Vert.Edges do
        if E == Edge then
            continue;
        end
        Other ← E.GetOther(Vert);
        if VFlags [Other.ID] then
            continue;
        end
        if Other == Edge.GetB() then
            return true;
        end
        VertQueue.Enqueue(Other);
        VFlags [Other.ID] ← true;
    end
    VertQueue.Pop();
end
return false;
```

Figure 4.24: This algorithm is used on an edge that has references to its vertices A and B, it tries to traverse the graph using BFS, it starts with A and if it can reach B the edge is part of a cycle.

4.6.3 Level Subdivision algorithm

The final puzzle generation algorithm uses the connectivity graph and acyclic edge information which was computed earlier to find a suitable location to divide the level by placing the gate, then it searches the graph to figure out which nodes belong to which sub-graph it also finds the side which is closest to the starter volume, to place the pressure plate such that the puzzle will be solve-able. The idea is to end up with a graph in which nodes represent a sub-graph of a part of

the level, which has the same neighbours as the sub-graph, the super-graph should also have a reference to the edge that connects the sub-graphs. The maintenance of all this data, when splitting the graph is what makes this algorithm a bit tricky. Pseudo-code for the algorithm can be found in Figure 4.26.
One can question the necessity of maintaining these data structures, when we could have just found the edge and then found the side which is closest to the starter volume to place the gate and pressure plate. However, these data structures provide a good way of representing the space and constraints of the puzzle in a structured manner, which is going to be useful later when evaluating the puzzle.
An example of the gate and pressure plate placement can be seen in Figure 4.25.

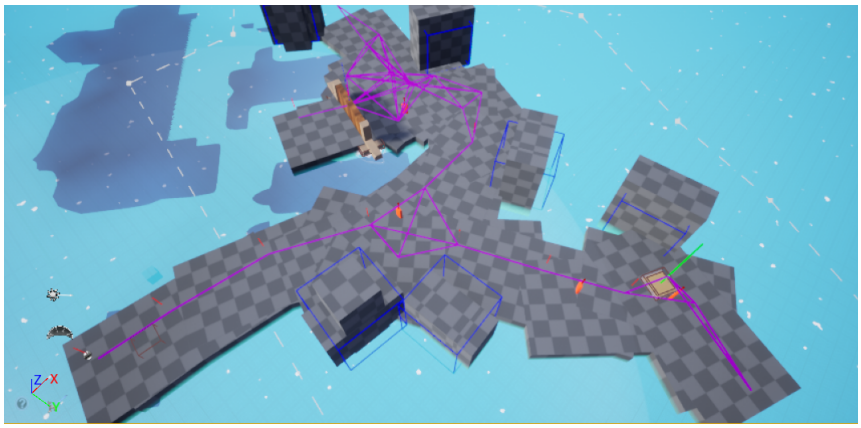


Figure 4.25: This image shows the connectivity graph which knows its acyclic edges, these edges are used to place the gate and the pressure plate. The image shows one of the better examples of the gate and pressure plate placements.

Algorithm 9: Graph Splitting Algorithm

input : The Edge at which we want to split
output: Updated data structures of the super-graphs & sub-graphs

```
AVertices ← (empty list);
BVertices ← (empty list);
AEdges ← (empty list);
BEdges ← (empty list);
VisitedCount ← 0;
VFlags ← (empty list);
for i ← 0 to TotalVertCount do
    VFlags.Add(false);
end
VFlags [SplitEdge.GetA().ID] ← true;
Populates AVertices, AEdges and tries to find the start in A;
VisitedCount ← BFS(SplitEdge.GetA(), VFlags, AVertices, AEdges);
if VisitedCount < TotalVertCount then
    Populates BVertices, BEdges and tries to find the start in B;
    VisitedCount ← BFS(SplitEdge.GetB(), VFlags, BVertices, BEdges);
end
if SuperGraph.Vetices.Num() > 0 then
    Initializes the SuperGraph & SubGraphs with the first split;
    InitSplitGraph(SuperGraph, SubGraphs, AVertices, AEdges, BVertices,
     BEdges, SplitEdge);
else
    Splits the Graphs further, updating the SuperGraph and SubGraphs
     accordingly;
    SplitGraph(SuperGraph, SubGraphs, AVertices, AEdges, BVertices, BEdges,
     SplitEdge);
end
```

Figure 4.26

4.7 Level Evaluation

The last part of the puzzle generation process is to evaluate the generated output, as also discussed in chapter 3, the evaluation process has additional responsibilities other than verifying that there exists a solution. namely to provide the player with a set of domino bricks that solve the level. However, the level evaluation, unfortunately, is the part of this project that received the least amount of attention and as such the latter part i.e providing the user with bricks was not solved. The

algorithm that was implemented for evaluating the level tries to solve the problem of finding a set of connections that is an optimal solution in terms of minimising the distance required to travel to complete the level. Now this distance only takes into account the functionality of bricks with no modifiers and as such should only be considered as a hint for the designer to provide an adequate amount of bricks to solve the level.
The algorithm that solves the level tries to find a tree that connects all of the pre-placed Domino bricks with the starter volume without introducing any cycles. An efficient solution in which data format matches this application, to this problem, was found in 1956 by Kruskskal[12]. Kruskal's algorithm finds a minimum spanning tree (MST), one important thing to note though is that the tree found, is not guaranteed to be the least cost minimum spanning tree (LC-MST) there exists. As such there might exist more optimal solutions to the problem, there does exist solutions for finding the LC-MST see[13] however for this application Kruskals algorithm do suffice as all we need is a good approximation. Kruskal's algorithm works by starting at a vertex, then we add all of its weighted edges to a candidate list, we pick the lowest cost edge and add it to our solution tree, the newly visited vertices edges should be added to the candidate list, however, if the edge terminates at a vertex we already have visited we ignore it. This is continued until all vertices are connected. The implementation of Kruskals algorithm can be seen inFigure 4.27. Now Kruskals algorithm requires a graph as input, the nature of this graph is very important as it ensures that the solution found is an optimal one. This graph is computed by utilising the earlier mentioned navigation mesh also seen in Figure 4.23, by creating a vertex at all of the pre-placed bricks and starter volumes position, we can then compute the navigation path from a vertex to all other vertices, then pick the 4 shortest paths and construct edges between those vertices. As we are constructing edges from all vertices to their 4 nearest neighbours in the navigation mesh domain we ensure that all vertices in the graph are connected which is a requirement for Kruskal's algorithm. A small note on loosely versus densely connected graphs is that the more dense a graph is connected the higher the probability of finding the LC-MST is, but this comes with a computational trade-off as the algorithm uses a sorting algorithm for the edges after newly visited vertex edges are added to the candidate list. An example of an MST solution in a constructed level can be seen in Figure 4.28.

Algorithm 10: Kruskals Algorithm

```
input : A Graph of (V,E) with all vertices connected
output: A set of edges representing a minimum spanning tree
Size ← Graph.Vertices.Num();
for i to Size do
    VFlags.Add(false);
end
VFlags [0] ← true;
MSTEdges ← (empty list);
Edgelist ← (empty list);
NewVert ← Graph.Vertices[0];
while MSTEdges.Num() < Graph.Vertices.Num() − 1 do
    for i to NewVert.Edges.Num() do
        Flags ← VFlags [NewVert.Edges[i].GetA ().ID] and VFlags
         [NewVert.Edges[i].GetB().ID];
        if ! Flags then
            Duplicate ← false;
            for Mst in MSTEdges do
                if Mst == NewVert.Edges[i] then
                    Duplicate ← true;
                end
            end
            if ! Duplicate then
                Edgelist.Add(NewVert.Edges[i]);
            end
        end
    end
    Edgelist.Sort(SortByWeight());
    MSTEdges.Add(Edgelist [0]);
    if VFlags [Edgelist [0].GetA().ID then
        NewVert = Edgelist [0].GetB();
    else
        NewVert = Edgelist [0].GetA();
    end
    VFlags [Edgelist [0].GetA().ID] ← true;
    VFlags [Edgelist [0].GetB().ID] ← true;
    Edgelist.RemoveAt(0);
end
return MSTEdges;
```

Figure 4.27: this projects implementation of Kruskals algorithm[12] used to find a set of edges that represent a minimum spanning tree.

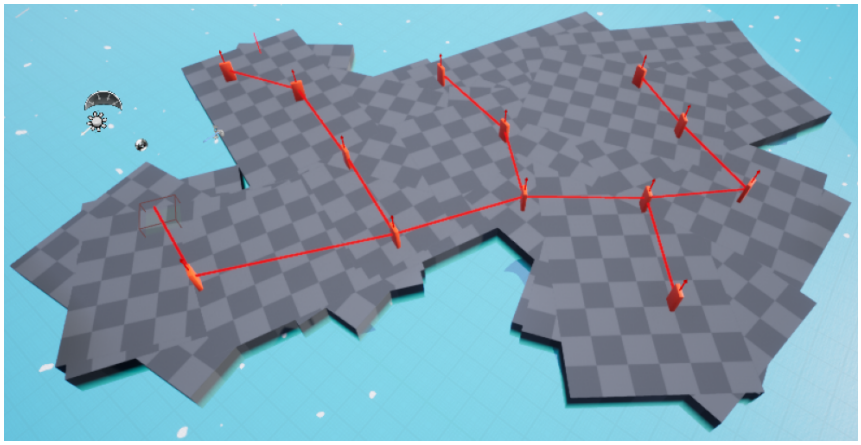


Figure 4.28: This image shows an example of Kruskals MST algorithm in use in an example level.

Chapter 5

Final Solution

In this chapter the final solution is showcased in the images seen in Figure 5.1 and Figure 5.2. The first image in Figure 5.2, shows a selected that was produced by the generator in unreal, the image is excluding the generated geometry that was generated by the HDAs. The second image in Figure 5.2 shows the data from the first image transformed by the HDAs. The generated result is a near completed level that, with some designer modification easily could make it into the final game. The ability to generate levels as shown in Figure 5.2 by the click of a button is immensely powerful and will greatly reduce the required man-hours to complete the game. This reduction in man-hours also as a result enables us to focus more time on the puzzles rather than constructing levels the traditional way.

Figure 5.1: This image shows a selected level that was produced by the generator, the level excludes the HDAs final generated geometry to showcase the building blocks the designer can modify to change the level.

Figure 5.2: This image shows the same selected level from Figure 5.1, but with the HDAs enabled, the image represents a level that with some designer modifications could be a candidate for the final game.

Chapter 6

Discussion

In this chapter I will present some reflections on the produced artefacts, some aspects of the generators was a success, while others lack features. with these reflections in mind, I will present some of the ideas and approaches that were not finished in time or has at the end of development dawned on me in terms of future development.

6.1 Successful Aspects

Great Tool that Provides Workflow improvements

The first successful aspect of the project was the whole workflow of the generation process, the modifiable BSP Brushes and curves, that updates the results of the HDAs in near real-time is an immensely powerful way of creating game content that allows the designer to make small changes to levels, with little to no effort and see the updates after some seconds of processing. The fact that the interplay between the Unreal based generator and the Houdini Digital Assets was a success that set up entire levels with the click of a button was a huge success. The level generator in it allows the designer to discover level configurations that present ideas of puzzle layouts, referring back to the reflections of Jonathan Blow on the development of Braid[2] that was presented in chapter 2, the level generator achieves what we set out for it to do.

Successful Use of Graph Theory

In terms of puzzle generation, one can argue that the produced artefacts in terms of their configurations are trivial, which is to a certain extent is true. However, the approach of using graph theory to model the problems turned out to be a successful way of analysing the generated levels. Here I want to highlight the acyclic edge detection, subdivision of a graph into two sub-graphs. Even though their application is currently trivial and Kruskal's MST, these methods represent an API that allows the programmer to easily create a random puzzle level. The methods represent the beginning of a toolbox that in the future could be used to create complex puzzle levels.

Understanding the Problem

The puzzle evaluation algorithm was a success even though it only solved a very small subset of the total problem. At the beginning of the project, I used a significant amount of time trying to model the actual placement of Domino bricks as a step towards simulating the level to verify if the level was solve-able. This process was a catalyst for reflections about the actual problem that the generator should try to solve. upon realising that the problem we were trying to solve was simply to evaluate if the level afforded a solution and to provide a set of Domino bricks to do so, the margin of error in terms of oddly oriented bricks and slightly misplaced entities was quite large. As any level produced by the generator would have to be hand-corrected/ manipulated by a designer anyways.

A Time Saver & Workflow Improvement

Even though the generator tool was not finished, it was still completed to a level where with some final polish, the development load of the game is reduced significantly as instead of focusing on creating 3d models for the levels and modifying those models when a change is needed the developer can produce more levels with greater quality.

6.2 Lacking Aspects

Shallow Generated Puzzles

As also mentioned in the positive aspects, the lack of puzzle elements makes to produced puzzles quite shallow in their complexity, it would be quite nice if we could achieve the same measure of exploration with the puzzle generation that we do with the level generator. The solution to this problem is not that far off, however, it also requires the game to implement more features that can be employed at random. These features would be environmental modifiers that alter the solution space by for example introduces a timing window in which something has to happen, this could be an automatically opening door, a rotating axle with beams attached to it, a combination of pressure plates that needs to be triggered in/order to open a gate, A pressure plate which opens a door when it is triggered but closes another. Now some of these ideas will be extremely hard to model in the evaluation step, for the physics-based interactions one could look at race car simulation problems in[16] they made a model to control the racecar using the A* algorithm, however, alternatives such as reinforcement learning could be used to solve the physics interaction aspect of the problem.

Evaluator Provided Bricks

Another low-hanging fruit that did not make it was the evaluator providing bricks for the generated level, at the very least we already computed the total path length needed to be travelled along the MST tree which could be translated into subdivisions plus some error that could provide the designer with an amount of non-modifier bricks, that roughly represents how many bricks is required to complete the level, from there the designer can then translate some of the provided bricks into ones with modifiers manually.

HDA Cook Duration

One more straightforward problem was the HDAs processing duration, this is apparent when they are spawned by the generator. It takes up to several minutes for the cooking process to be completed, which leaves the designer losing focus and makes it impossible to enter a state of flow. The solution is to optimize the HDAs and there is a lot of low hanging fruits for optimizing in terms of unneeded computations, duplicate computations and better alternatives for algorithms with

bad complexity.

The bridge between UE4 and Houdini Engine was a success, however, it also presented a lot of problems that needed and still needs to be solved. Working with released open-source software, it is possible to modify the engine and rebuild it. However in the act of doing so, when UE4 or Houdini updates their code, manual work would be needed to merge to changes that we made with the new plugin. As such it would be nice to try and contact developers of each engine and request the APIs to be modified/ extended to better fit the requirements of the project, as others who might walk the same path, would have a much easier time bridging the two tools.

6.3 Future Work

Designer User Experience

In chapter 2, we briefly touched on the degree and dimension of control, which was to some degree put aside in this project, in Figure 4.1, in chapter 4, a UI was presented that contains a few variables that modify the generated level, however, currently, none exists for the puzzle generation. A big focus for the future development of this project is going to be focusing on making the interaction with the tool as easy as possible. One way of achieving this is by better communicating what the individual parameters do, currently, they have a name that is heavily influenced by their use in the code, though in the future a better description for each of them will be needed. Another point to make here is to find sensible bounds for each of the parameters to constrain the generation to behave in a way that is intended. Currently, it is possible to insert values in some of the parameters that will entirely break the generation process. In[5], they also present the idea of representing the puzzle parameters as a feature vector, where each element should be representative of something tangible within the puzzle eg. producing many pressure plates & gates that makes the puzzle more difficult in the combinatorics domain, the vector should be normalized meaning a value of 1 in the combinatorics domain would produce very hard problems and a value of 0 would ignore this aspect. Having such a feature vector would make it easy for the designer to experiment within a generated level, to produce the type of puzzle that fits the best.

Algorithms

The use of graph theory opened a world of a whole new set of solutions and problems for us to explore in the context of Puzzle PCG. In this little section, we will present algorithms that show promise to solve or some of the remaining problems or improve existing ones.

Re-representing the Connectivity Graph

The connectivity graph that was presented in chapter 4, represented to some extent how the level was connected, however, the skewed distribution of nodes, where branches have grown on top of each other, resulted in a graph that reduced the number of possibilities in terms of gate placement. With the same methodological approach but with different algorithms, this connectivity graph might be more representative of the actual level. If instead we re-sampled the navigation mesh Figure 4.23, with points scattered in an evenly distributed grid, with a modifiable distance between them and construct a graph using the same algorithm to construct the connectivity graph we would have a much more evenly distributed graph. However, this would render the current acyclic edge approach obsolete, as the graph would be more densely connected as such we would have to find an alternative algorithm to accomplish the same thing.
Another useful feature of the re-represented connectivity graph is its potential to create informed distributions of the pre-placed Domino bricks where the error of oddly oriented bricks could be used by computing the average direction of the graph at randomly sampled vertices.

Maximum Flow Problem

One solution to detecting bottlenecks in the level that pose a good way of finding natural gate locations could be the maximum flow problem. The maximum flow problem that posed in 1955 by T.E. Harris and F.S. Ross[17], in the context of locomotive transportation bottlenecks. A solution to the maximum flow problem was not soon after found by Ford, L., & Fulkerson, D[6]. With the idea of representing the maximum flow that can pass through a network, one can imagine a great method of finding the bottlenecks. If we could compute flow weights between selected nodes, we could find several candidates for placements, which is an improvement from the current method.

Delay Constrained MST

Another dimension of the puzzle that was discussed inchapter 2, is the time domain. The time-domain presents a new dimension in terms of the evaluation which was proved to be NP-Complete by[15]. However, they also presented heuristics based on Prim's algorithm that could find a solution that is close to optimal. Prim's algorithm uses a very similar approach to Kruskal's algorithm. Now the delay constraint is a value that can be assigned to every edge, but in our situation, this would have to be done after the initial MST has been found, as the delay value would only be known for a certain solution, then we could modify the MST with the delay constraint in mind. Now it's important to note that it might not work at all but it seems like a possible solution to the problem.

Physarum Shortest path computation

At the very end of this project, I discovered an article that shows an algorithm that presents an alternative solution to the path-finding problem. This algorithm is based on the "natural algorithm" that dictates the growth of the fungi Physarum Polycephalum[3]. The growth of the fungi can be seen in a video[23], this algorithm presents an interesting alternative to many of the current approaches that were used in this project, potentially this could be used before the MST algorithm, where the fungi approach models a unique optimisation that can split an edge heading towards two neighbouring nodes, thus creating a new node to optimise the graph instead of just connecting the nodes. This approach could potentially produce a more optimal MST.

Answer Set Programming

Finally, the placement of environmental modifiers layout and the layout of the pre-placed bricks could be solved by using ASP which was also done in Anza Island[4]. ASP was set aside for this project as it requires the game to be complete to generate results, which is an important point about the puzzle generation, it is only something that can be modelled after the game has been discovered and described by a set of rules that must be well defined. This was not entirely the case for this project as we are still in the early development phase.

Chapter 7

Conclusion

This project set out to explore the creation of a procedural puzzle generation tool that allows the designer full control of manipulating its generated artefacts at each stage in its generation process. To achieve this, the state of the art game development tools Unreal Engine 4 & Houdini Engine was used as a platform. The generation process was divided into three stages, level generation, puzzle generation and puzzle evaluation. Each stage presented a unique set of problems that were tackled with varying success. A tree generation algorithm that created a level layout of BSP volumes, a BSP house generation algorithm and a curve generation algorithm was used to supply data to Houdini Digital Assets that would generate the final geometry that makes up the levels. Upon the generated geometry a puzzle creation algorithm was used to create a layout of game elements. Finally, the level could to some extent be evaluated for solvability. The puzzle generation and evaluation still needed some work to be considered done. Although no evaluation was conducted the level generation process of the tool can still be regarded as a success as it reduces development time and the resulting quality significantly. Finally, we presented existing algorithms and methods that could be used for the further development of the project.

Bibliography

[1] Paul Ambrosiussen. *Paul Ambrosiussen*. https://www.linkedin.com/in/paulambrosiussen/.

[2] Johnathan Blow. *How Jonathan Blow Designs a Puzzle*. https://www.youtube.com/watch?v=2zK8ItePe3Y. 2016.

[3] Vincenzo Bonifaci, Kurt Mehlhorn, and Girish Varma. "Physarum can compute shortest paths". In: *Journal of Theoretical Biology* 309 (2012), pp. 121–133. ISSN: 0022-5193. DOI: https://doi.org/10.1016/j.jtbi.2012.06.017. URL: https://www.sciencedirect.com/science/article/pii/S0022519312003049.

[4] Kate Compton, Adam Smith, and Michael Mateas. "Anza Island: Novel Gameplay Using ASP". In: PCG'12. Raleigh, NC, USA: Association for Computing Machinery, 2012, 1–4. ISBN: 9781450314473. DOI: 10.1145/2538528.2538539. URL: https://doi-org.zorac.aub.aau.dk/10.1145/2538528.2538539.

[5] Barbara De Kegel and Mads Haahr. "Procedural Puzzle Generation: A Survey". In: *IEEE Transactions on Games* 12.1 (2020), pp. 21–40. DOI: 10.1109/TG.2019.2917792.

[6] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), 399–404. DOI: 10.4153/CJM-1956-045-5.

[7] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. "On Visible Surface Generation by a Priori Tree Structures". In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '80. Seattle, Washington, USA: Association for Computing Machinery, 1980, 124–133. ISBN: 0897910214. DOI: 10.1145/800250.807481. URL: https://doi.org/10.1145/800250.807481.

[8] Epic Games. *Brush System Documentation in UE4*. https://docs.unrealengine.com/en-US/Basics/Actors/Brushes/index.html. 2004.

[9] Epic Games. *Unreal Engine 4*. https://www.unrealengine.com/en-US/. 2004.

[10] Hello Games. *No Mans Sky*. https://www.nomanssky.com/. 2016.

[11] Studio Ghibli. *Colour palette used by Studio Ghibli*. http://designmadeinjapan.com/magazine/graphic-design/the-rich-colors-of-studio-ghibli/. 1985.

[12] R. L. Graham and Pavol Hell. "On the History of the Minimum Spanning Tree Problem". In: *IEEE Ann. Hist. Comput.* 7.1 (Jan. 1985), 43–57. ISSN: 1058-6180. DOI: 10.1109/MAHC.1985.10011. URL: https://doi.org/10.1109/MAHC.1985.10011.

[13] M.R. Hassan. "An efficient method to solve least-cost minimum spanning tree (LC-MST) problem". In: *Journal of King Saud University - Computer and Information Sciences* 24.2 (2012), pp. 101–105. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2011.12.001. URL: https://www.sciencedirect.com/science/article/pii/S1319157811000401.

[14] Mojang. *Minecraft*. https://www.minecraft.net/en-us. 2009.

[15] H.F. Salama, D.s Reeves, and Yannis Viniotis. "The delay-constrained minimum spanning tree problem". In: Aug. 1997, pp. 699–703. ISBN: 0-8186-7852-6. DOI: 10.1109/ISCC.1997.616089.

[16] Yoppy Sazaki, Anggina Primanita, and Muhammad Syahroyni. "Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A". In: July 2017, pp. 164–169. DOI: 10.1109/ICWT.2017.8284160.

[17] Alexander Schrijver. "On the history of the transportation and the maximum flow problems". eng. In: *Mathematical programming* 91.3 (2002), pp. 437–445. ISSN: 0025-5610.

[18] Robert A Schumacker et al. *STUDY FOR APPLYING COMPUTER-GENERATED IMAGES TO VISUAL SIMULATION*. eng. 1969.

[19] Mohammad Shaker et al. "Automatic generation and analysis of physics-based puzzle games". In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. 2013, pp. 1–8. DOI: 10.1109/CIG.2013.6633633.

[20] SideFX. *Houdini*. https://www.sidefx.com/company/about-sidefx/. 1987.

[21] Adam Smith et al. "A case study of expressively constrainable level design automation tools for a puzzle game". In: *Foundations of Digital Games 2012, FDG 2012 - Conference Program* (May 2012). DOI: 10.1145/2282338.2282370.

[22] Wube Software. *Factorio*. https://www.factorio.com/. 2015.

[23] Seiji Takagi. *Physarum Growth*. https://www.youtube.com/watch?v=BZUQQmcR5-g. 2010.