Faculty of Computing and Digital Technologies

Staffordshire University

**BACHELOR THESIS**

Christos – Emmanouil Anastasiou

**Efficient simulation of destructible environments in video games**

A project submitted in partial fulfilment of the award of the degree of BSc (Hons) Computer Games programming from Staffordshire University

Supervisor of the bachelor thesis: Bob Hobbs

May 2020

Title: Efficient Simulation of destructible environments in video games

Author: Christos – Emmanouil Anastasiou

Supervisor: Bob Hobbs

Abstract: Investigation of currently popular methods for implementing destructible environments in a computer game and testing the performance of one of the methods. The method chosen involves the qualities of voxels and how their volumetric nature can be used to generate destructible environments.

Keywords: voxel game engine, destroyable environments, destructibility, geometry preparation, runtime destruction.

Contents

[CHAPTER 1 1](#_Toc40412150)

[1.1 Introduction 1](#_Toc40412151)

[1.2 Objective 2](#_Toc40412152)

[1.3 Thesis structure 2](#_Toc40412153)

[CHAPTER 2 3](#_Toc40412154)

[2.1 The importance of project planning 3](#_Toc40412155)

[2.2 Selection of project planning methodology 4](#_Toc40412156)

[2.3 Tools 6](#_Toc40412157)

[2.3.1 Trello 6](#_Toc40412158)

[2.3.2 Team Gantt 7](#_Toc40412159)

[CHAPTER 3 8](#_Toc40412160)

[3.1 Overview of techniques 8](#_Toc40412161)

[3.1.1 Geometry preparation 9](#_Toc40412162)

[3.1.2 Real time destruction methods 12](#_Toc40412163)

[3.1.3 Terrain 13](#_Toc40412164)

[3.2 Definition of the project specification 17](#_Toc40412165)

[Outline of proposed iterations 18](#_Toc40412166)

[CHAPTER 4 20](#_Toc40412167)

[4.1 Iteration 1 – Program structure and design 20](#_Toc40412169)

[4.1.1 Selection of Application Programmable Interface (API) 20](#_Toc40412170)

[4.1.2 Product backlog, MoSCoW 21](#_Toc40412171)

[4.1.3 Class diagram 22](#_Toc40412172)

[4.1.4 Test objectives 22](#_Toc40412174)

[4.1.5 Review 23](#_Toc40412175)

[4.1.6 Retrospective 23](#_Toc40412176)

[4.2 ITERATION 2 – framework set up 23](#_Toc40412177)

[4.2.1 Product backlog, MoSCoW 23](#_Toc40412178)

[4.2.2 Class diagram 24](#_Toc40412179)

[4.2.3 Test objectives 25](#_Toc40412180)

[4.2.4 Review 26](#_Toc40412181)

[4.2.5 Retrospective 26](#_Toc40412182)

[4.3 ITERATION 3 – Voxel placement 27](#_Toc40412183)

[4.3.1 Product backlog, MoSCoW 27](#_Toc40412184)

[4.3.2 Class diagram 29](#_Toc40412185)

[4.3.3 Test objectives 30](#_Toc40412186)

[4.3.4 Review 30](#_Toc40412187)

[4.3.5 Retrospective 30](#_Toc40412188)

[4.4 ITERATION 4 – Voxel organisation 31](#_Toc40412189)

[4.4.1 Product backlog, MoSCoW 32](#_Toc40412190)

[4.4.2 Class diagram 32](#_Toc40412191)

[4.4.3 Test objectives 34](#_Toc40412192)

[4.4.4 Review 35](#_Toc40412193)

[4.4.5 Retrospective 35](#_Toc40412194)

[CHAPTER 5 36](#_Toc40412195)

[5.1 Evaluating the success of the project 36](#_Toc40412196)

[5.1.1 Experience gained 36](#_Toc40412197)

[5.1.2 Final conclusion 37](#_Toc40412198)

[5.1.3 Future works 39](#_Toc40412199)

[References 40](#_Toc40412200)

# 

## Introduction

Computer games have broken new ground on rendering and animation in the past decade, achieving new levels of realism and believability that contribute to the overall immersion [1] [2]. That illusion, however, can easily be dispelled when the player finds a chain-link fence that is impervious to anything from a machine gun to a tank. Destructible environments refer to the player’s ability to destroy their surroundings, thus knowing that they can use the environment to their advantage [3]. For example, the player can explode a barrel to eliminate enemies that are standing near it as seen in most shooter games or demolish part of a building to gain tactical advantage over other players as seen in Battlefield series. Some early games offered minor destructibility in their levels such as demolishable covers, breakable crates or destructible CCTV cameras such as the game Perfect Dark that also featured specific walls that had “weak” points that could be shot at and destroyed to open paths to subsequent levels.

Breakable objects made their first appearance in Gun Fight (1975) where two players engage in a gun fight while standing opposite to one another and being separated by cactuses that absorb gunshots and disappear partially depending to where the bullet was landed. Not long after, Space Invaders (1978) was released where the player was being protected by simple bricks that were slowly diminishing after each shot. Asteroids was released a year later in 1979 [4] where the player had to shoot flying asteroids to break them into smaller pieces. Rampage (1986) was one of the earliest classic arcade games that featured fully destructible environments where players were taking control of a giant ape with the objective of taking down apartment buildings and cause mayhem in the city.

At the time where games were transitioning from 2D to 3D graphics in the mid-nineties, games such as Magic Carpet (1994) was one of the first 3D games to feature terrain deformation with the use of height maps, the terrain was being modified by altering the height of a given point on the map [5]. Some later games such as Worms (1995) featured fully destructible terrain where the player would fight other worms with a large arsenal of weapons in a 2D setting.

By the year 2000, specialised pieces of hardware or graphics accelerator cards currently known as graphics processing units (GPU) brought games that took destructibility to the next level. The Red Faction series was primarily focused on the destructibility of its environments and buildings, more details on section 3.1.2. The Crysis series was developed on Crytek’s proprietary engine CryEngine [34] and featured destroyable objects using various techniques that are discussed in detail in later sections such as jointed breakable objects, object replacement and Boolean operations. Far Cry 2 (2008) featured a different form of destructibility and that was dry grass that when burned it would spread out giving the player an interesting addition into his tactical options. In the modern age of gaming, large studios such as DICE and Ubisoft Montreal invested valuable time and effort in developing destruction systems into their proprietary engines – Frostbite [7] and AnvilNext [8] respectively - to import into their online, multiplayer shooter games. Current releases include Remedy studio’s Control that is experimenting with realistic small-scale destroyable objects i.e. all furniture and solid material inside a building can be broken down realistically. Also, Noita (2019) currently in early access and developed by Nolla Games is a 2D procedurally generated world where every pixel is physically simulated [35].

Destruction in video games have played its important role in enhancing physics and tactical manoeuvring, two of the five types of game mechanics listed in Game Mechanics: Advanced Game Design by Ernest Adams and Joris Dormans. Producing non-scripted destructible environments that are most importantly fun and simulated realistically while also protecting the player from themselves from breaking the game has been a tough challenge for developers and designers in the past. Many game companies have acknowledged over the years that there is a rise of demand on physically accurate destruction, as it can prove to be a good addition to gameplay while going alongside with the game’s context [9].

Current advances in the field aim to accomplish more realistic; interactable game worlds by innovating and refining several available approaches used for 2D and 3D environments. Although latest graphical hardware components are becoming increasingly powerful [10], implementing a highly complex physical simulation introduces a development hurdle: the more accurate a simulation is the more impact it has on the system. Taking into consideration the real-time constraints that are required for a computer game to perform sufficiently by achieving the highest and most consistent frame rate possible, game developers are left with only one choice - to deduct some of the realism by creating simple and compact scenes but also by disregarding inessential aspects of physical simulation.

## Objective

There has been a lack of dynamic destruction systems on many games in the past. Canned animations are pre-rendered animations and not actual simulations that are triggered on certain events. Their use has proven to be non-beneficial to gameplay but only as visual effects.

The purpose of this thesis is to closely examine currently available techniques found from research that are used to simulate destructible environments. One of the techniques is chosen for testing its performance in a suitable computer games scenario. An artefact of this technique is being produced and used in a game environment.

## Thesis Structure

**Chapter 2: Importance of project planning**

This chapter describes the importance of project planning in software development and selects the appropriate methodology for the requirements of this thesis.

**Chapter 3: Overview of techniques**

This chapter describes and compares various methods used to simulate destructible environments during development and in real-time. One method is selected for implementation using the DirectX11 API.

**Chapter 4: Design and Implementation**

This chapter describes the design and implementation of the method selected in previous chapter.

# 

## The importance of project planning

The moment after a new project has come up and been approved, it is tempting to hurry and start carrying out the work immediately as there are people to talk to, things to purchase and a long list of tasks to write. That is the wrong approach to take even how tempting it first seems to be [11]. Before diving into the doing of the project, it is a smart move to take a step back to have a better view of the bigger picture of everything that needs to be produced and the way that they are going to be developed. For this to happen, a plan of the of the upcoming work is necessary. This will surely slow down the start of the actual implementation of the project, it will however save a significant amount of time in the long run due to how convenient it will be to keep track of the project’s progress but also have a clear image of future tasks.

A project plan streamlines the operation and prevents many probable incidences of misunderstanding and confusion for whoever is involved [12]. For instance, an experienced project manager will think ahead and assign the right tasks to the right people for the right dates. As a result, not having to worry if the workers are available for those dates as the PM already arranged for them to be. Another example is that the project manager will have already planned the financial paperwork thus not having to worry about paying the invoices after the job has concluded. Project planning consists of defining the processes that are crucial for determining the project’s objectives and facilitating the structure to enable these processes to come to fruition [13]. The structure represents the various procedures required for a successful project such as budgeting estimation, human resources, quality measures, deliverables etc. The outcome of that process is a project plan. A set of documents that explains what is going to be done and how.

Project Management Body of Knowledge guide (PMBOK) 2008 edition provides a definition of project management as “application of knowledge, skills, tools and techniques to project activities to meet project requirements. Project management is accomplished through the application and integration of the 42 logically grouped project management processes of initiating, planning, executing, monitoring and controlling, and closing” (*PMBOK guide, Project Management Institute, 2008, p.6*). What the PMBOK guide fails to mention is that there are other activities that are involved in project management such as handling political matters, ensuring that team members communicate with each other effectively and perform at the required level. (*James P. LEWIS, 2011*) [11].

Project planning is often confused with project schedule. That is a list of tasks and dates that specify what needs to be done and when. This tool will certainly come in use but is only a piece of the toolbox.

## Selection of project planning methodology

Jason Charvat (2003) in his book Project Management Methodologies, he defines project management methodology as follows: “A methodology is a set of guidelines or principles that can be tailored and applied to a specific situation. In a project environment, these guidelines might be a list of things to do. A methodology could also be a specific approach, templates, forms, and even checklists used over the project life cycle.” In other words, countless processes guide project managers throughout the project but also provide the steps for carrying out the tasks and running the operation smoothly. There are various project methodologies available that are currently in use but the most popular are Agile; Kanban; Waterfall; Lean; Six Sigma and Project Management Body of Knowledge guide (PMBOK) [14].

Selecting the appropriate planning methodology for the project is crucial as it determines not only the type of workflow that is going to be structured but also the flexibility and freedom needed for a healthy software development cycle (Charvat, 2003).

When selecting a methodology, it is best to first make a list of the criteria that the methodology needs to meet as this will later assist the searching process by narrowing down the options. It is worth considering that:

1. This is a solo project with research purposes and not with specific deliverables provided by a client.
2. It involves the development of software. Therefore, project planning methodologies designed for software development will only be taken into account.
3. The selected approach will work as a framework for managing workload and scheduling tasks while controlling the procedure of development by providing documentation for monitoring progress but most importantly allowing adjustments when required. Tweaking and fine tuning are crucial for improving the prioritization of tasks and ways of approaching unsolved problems resulting to additional planning as the project moves on to later stages of development.

For these reasons, **an Agile approach** appears to be the most suitable approach for the precise nature of this project. Secondly, the benefits of agile methodology are frequent iterations, continuous feedback and overall incremental development that encourages responding to change over following a plan (*Schwaber and Sutherland, 2018)*.

However, although there are many Agile approaches available, not all of them fit for this specific circumstance. The application of the Agile methodology that is being adopted is **Scrum** as it is designed for a small team of people thus making it easier to be modified for one-person operation [15]. Besides, Scrum is lightweight, simple to understand and widely used in the software development industry while also fitted for research purposes according to “The Scrum Guide” [16] that was crafted by Ken Schwaber and Jeff Sutherland, the creators of Scrum.

Alex Andrews’s article “Scrum Of One: How to Bring Scrum into your One-Person Operation” [17] was used as a reference and guidance for this plan. The author starts off by focusing on the core principles of Scrum rather than the operations that are designed for a team. The first principle states that it is essential to share with other people the state of the product after each iteration as it is notoriously easy to miss the right degree of importance of certain tasks and waste precious time on features that were never needed or wanted. The second principle declares that there should be short term goals that measure the level of productivity at the end of each week. It is important to quantify the progress, so it is easier to track and optimize it. The third principle refers to the ability of self-reflection and review of processes and progress in a regular basis. These principles correspond to the criteria mentioned above.

The chosen methodology operates on several rules comprised of events, tools and roles and the relationship between them. What follows is a trimmed version of traditional Scrum:

**Sprint**: Development is broken down into several iterative cycles also known as sprints. Sprints usually last from two weeks to up to 30 days and for that specified period, the development team focuses on a clear and detailed goal such as adding a new feature or fixing a collection of bugs (Andrews, 2017). A two-week sprint plan has been decided on to allocate enough time for a meaningful set of tasks to complete without having more than enough time to get carried away with insignificant tasks.

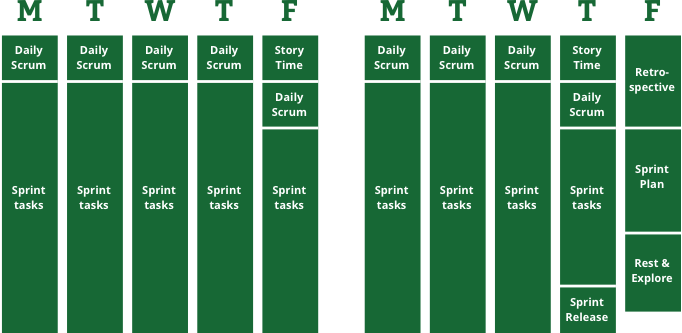


Figure 2.1: Alex Andrew’s Scrum of one: a two-week sprint plan broken down to reflect each day’s assigned sprint component.

**Sprint plan/Product backlog:** It Consists of a list of sprint goals that are broken down into smaller sprint tasks that are then considered for the next sprint. A technique Alex used is that each sprint task is associated with an arbitrary number that defines the rough estimated time of completion. For instance, one-point tasks should take on average half as long as two-point tasks. The numbers being used are 1,2,3,5 and 8. The numbers 5 and 8 are used to add a distinct value to the effort needed to complete a bigger task. if a certain task has been employed numerus times before, the chances are it will take less time thus reducing the number of task points regardless of its complexity. Similarly, an unfamiliar task will increase the number of task points even if it is simple. After including all the task points for the upcoming sprint, the next step is to add them all up. Consistency is the utmost importance here as the numbers are used to measure the productivity levels at the end of each sprint and compare the number to previous iterations.

**Daily scrum (5 minutes)**: will take place at the beginning of each working day. The daily scrum will consist of a quick update of yesterday’s progress, obstacles that blocked uncompleted tasks and today’s goals.

**Story time (30 - 45 minutes):** will take place at the end of each week and will allow time for shortly pausing development and having a look at the big-picture tasks. Also known as product backlog. Adding new tasks and reviewing the old tasks is equally as important.

**Task board:** comprises of a board of TODO, DOING and DONE categorised tasks. The tool selected for documenting the task board is Trello, an online tool, described below.

**Weekly sprint review:** Weekly meeting with supervisor that involves weekly update and discussion on obstacles found along the way and how to overcome them.

**Retrospective (~2 hours):** At the end of each week, self-reflection takes place on the week that went past by asking questions to detect what worked well and what could have been done better.

**Sprint release**: sharing the current state of the artefact with other people is important for receiving feedback as previously mentioned from one of the three principles. This will take place at the end of each sprint.

## Tools

### Trello

Trello is a free online project management tool designed for creating task boards. The task boards produced can be archived and accessed from any device connected to the internet. Thus, providing a synchronised platform for being able to develop in. A board is where the management for each project takes place. Each board is made up of a series of lists and each list contains several items that represent each task. These items are called cards and the idea of Trello is that as the project progresses, these cards can dynamically move – starting from the left - to subsequent lists that represent the next stages of the project. Trello is a visual representation of the project’s progress. The free version of Trello allows for a single addon.

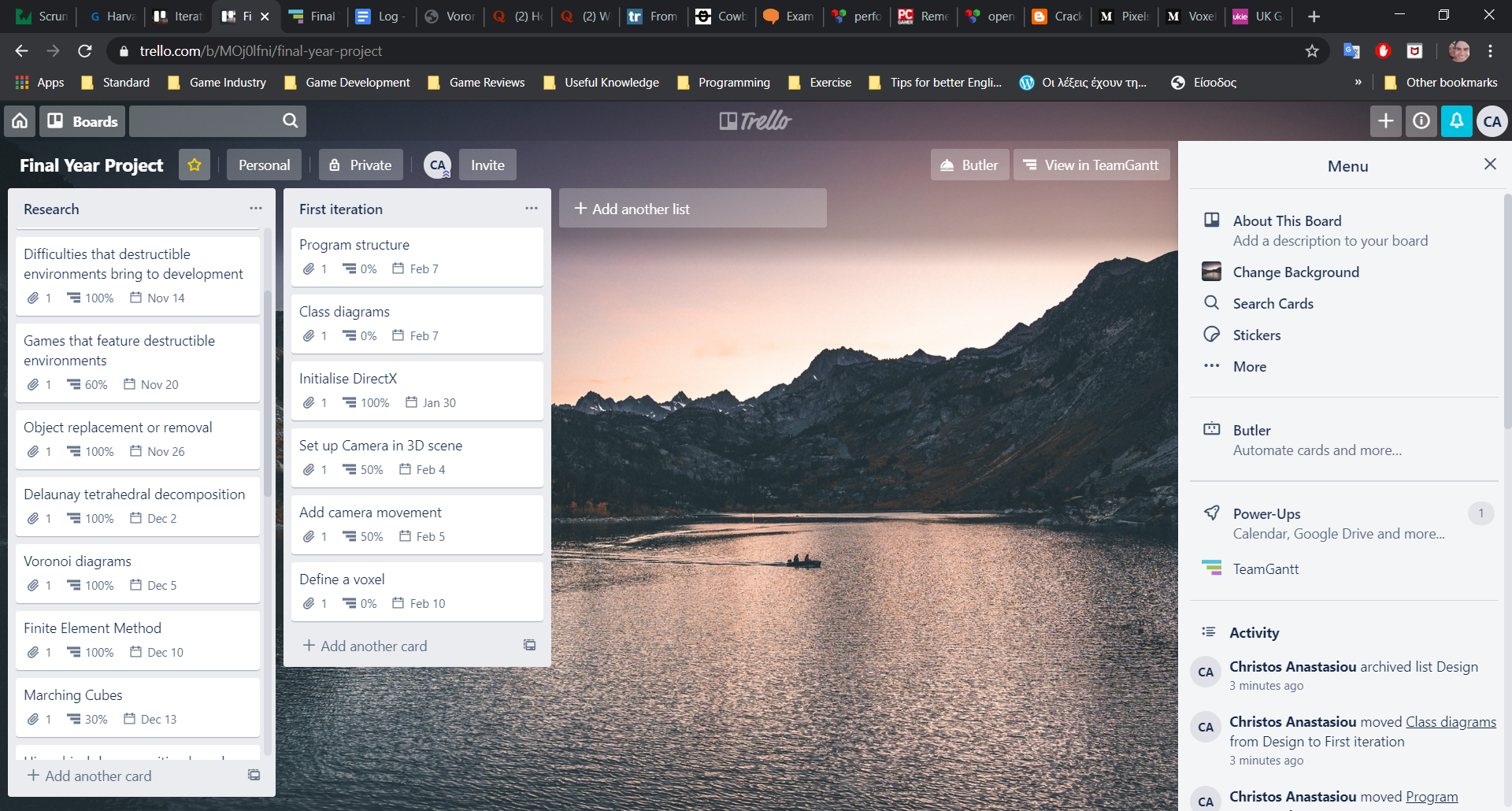


Figure 2.2: Visual representation of the project using cards and lists in a Trello board.

### Team Gantt

Team Gantt is an online software for producing dynamic Gantt charts and keep track of tasks for a project. It can be synchronised with a Trello board. Hence, breaking down and updating the tasks dynamically but also setting a deadline that can be represented visually.

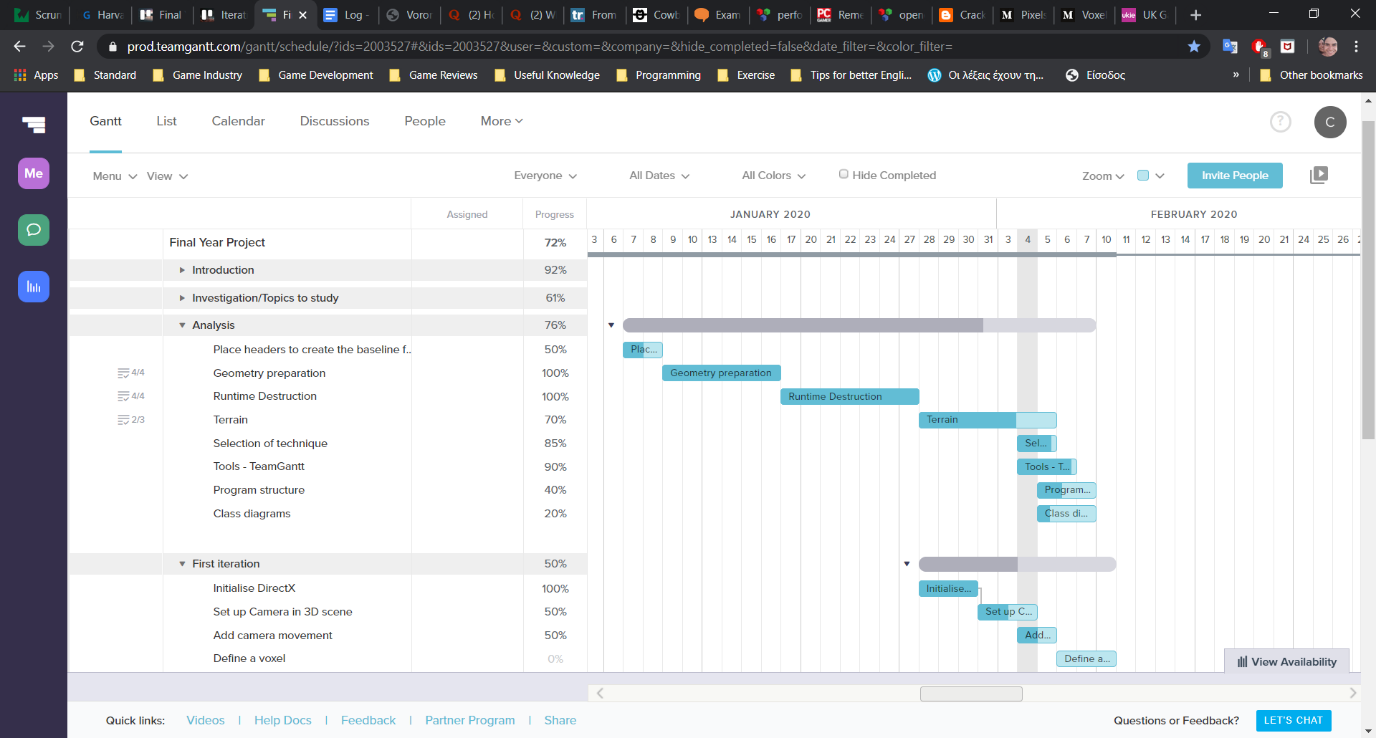


Figure 2.3: Example of Team Gantt. Milestones are broken down into smaller tasks that are represented visually based on the date that each task is set to be worked on. Some tasks can be dependent of other tasks, this can be represented with the line starting at the end of one task and connecting to the start of the next task thus blocking it from initiating simultaneously.

# 

## Overview of techniques

In this chapter, a brief overview of the destructible environment techniques used in various games developed in the last 40 years will be studied. This will assist in the development of a physically accurate and dynamic destroyable environment, thus avoiding reinventing the wheel.

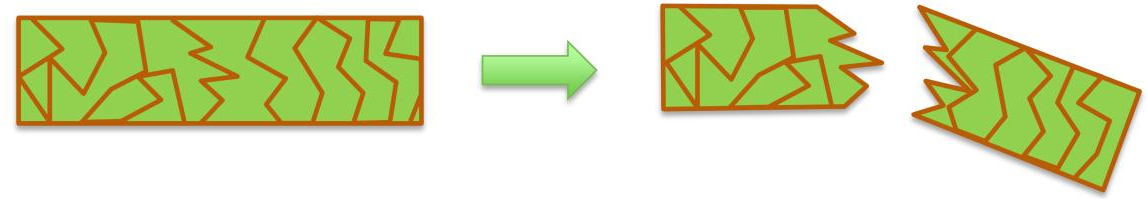
It is worth noting that most modern 3D video games use the term destructible environment as a reference to destroyable objects, such as any building, barrel, crate, tree or any other game object that appear in the game world. This excludes terrains, distant geometry such as sky-boxes and game characters as terrain deformation was not featured. In the simplified version of 2D view however, several games included the terrain when referring to destructible environments. As we can see in games such as Scorched Earth (1991) [18], Worms (1995) [19], Broforce (2015) [20] and Noita (2019) [21], removal is based on pixels rather than whole objects.

The following comparisons are primarily based upon AMD physics engineer Erwin Coumans’s presentation from SIGGRAPH 2011 [23] where he spoke about current solutions and research on breakable game assets. He broke the destruction process into two categories:

* Geometry preparation during development



* Runtime destruction methods



|  |  |
| --- | --- |
| Geometry Preparation | Runtime Destruction |
| Voronoi shatter, slicing | Canned animation |
| Boolean operations | Real-Time Booleans |
| Convex decomposition | Finite Element Method |
| Delaunay tetrahedral decomposition | Jointed Breakable Objects |

Table 3.1: Current methods used for geometry preparation and runtime destruction as described by Erwin.

### Geometry preparation

“A fracture is the separation of an object or material into two or more pieces under the action of stress.” [30]

Geometry preparation is a process that the artist is responsible for when creating destroyable objects. The artist processes 3D geometry through 3D modelling tools such as 3ds Max, Maya or Blender so that it produces fragments. The assets are fractured before are placed in the game scene or they replace their simpler counterparts during runtime, similarly to object replacement.

#### Object replacement or deactivation

Swapping an object with a more damaged version of itself or removing it completely was one of the earliest approaches to simulate breakable objects in games (see Figure 3.1). In fact, this method is still being chosen and widely used to this day due to its simplicity and undemanding implementation. It depends on a set pre-created models so the precise point of collision on the model is not taken into consideration. After the object has taken damage, it is usually followed by a particle effect such as smoke or dust to hide the object replacement (see Figure 3.2). This approach provides the highest degree of artistic control but can be costly in terms of man hours. It can still produce desirable results in spite of its limitations.



Figure 3.1: A set of pre-made models get swapped between them based on the amount of damage the door has taken regardless the points that the bullets have landed on. This example can be found in Counter Strike: Global Offensive (Valve,2012). [24], [5]



Figure 3.2: The crate is removed in a single frame but hidden behind particle effects such as spawned flying wooden pieces and glow. This example can be found in Crash Bandicoot N. Sane Trilogy (Vicarious Visions, 2017).

#### Voronoi shattering

Cutting a 3D model into smaller parts using Voronoi slicing is a popular way of simulating solid materials such as concrete, minerals and stone. It is a sweep line method based on Voronoi diagrams. A Voronoi diagram is a method of dividing the space that processes a group of points and generates an equal number of convex regions called cells. Fortune’s algorithm is a known algorithm for generating a Voronoi diagram from a selection of randomly placed points in a plane in O (n log n) worst-case running time and O(n) space [25].

Based on Hugo Ledoux’s paper [27], Voronoi shattering is produced in the following way:

First the polygonal geometry is sampled to seek for a set of randomly placed points called seeds. Then, for each seed is created a group of planes that result in their matching Voronoi cells. For each of these cells, the mesh is cut causing a hollow that is filled in with newly generated faces. A collection of fragments is the result of this procedure. For this algorithm to work, the parameters need to be a closed mesh and enclosed points.

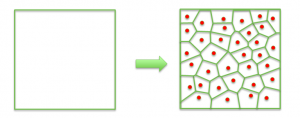


Figure 3.3: Voronoi Shattering, [23], [26]

#### Delaunay tetrahedral decomposition

A 3D mesh can be sub-divided into a set of tetrahedral elements following the process of tetrahedralization utilizing Delaunay Triangulation. This is done by given points P that are the mesh’s vertices and a collection of points inside its volume. In order for a tetrahedral mesh DT(P) to be generated, it needs to meet the following circumstance: “no point in P is inside the circumscribed sphere of any tetrahedra in DT(P)” [28]

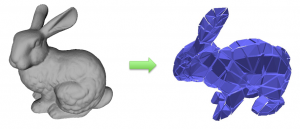
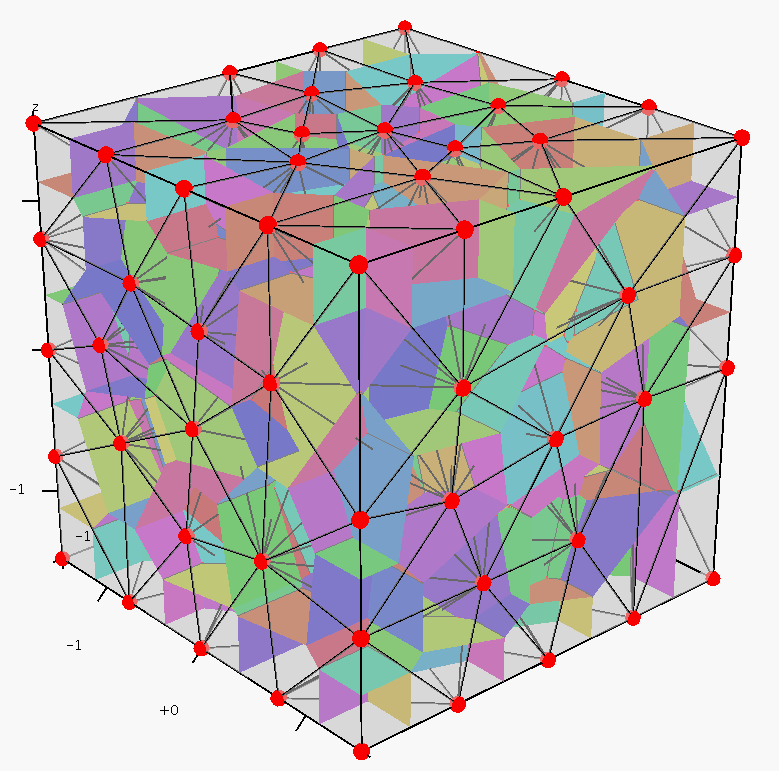
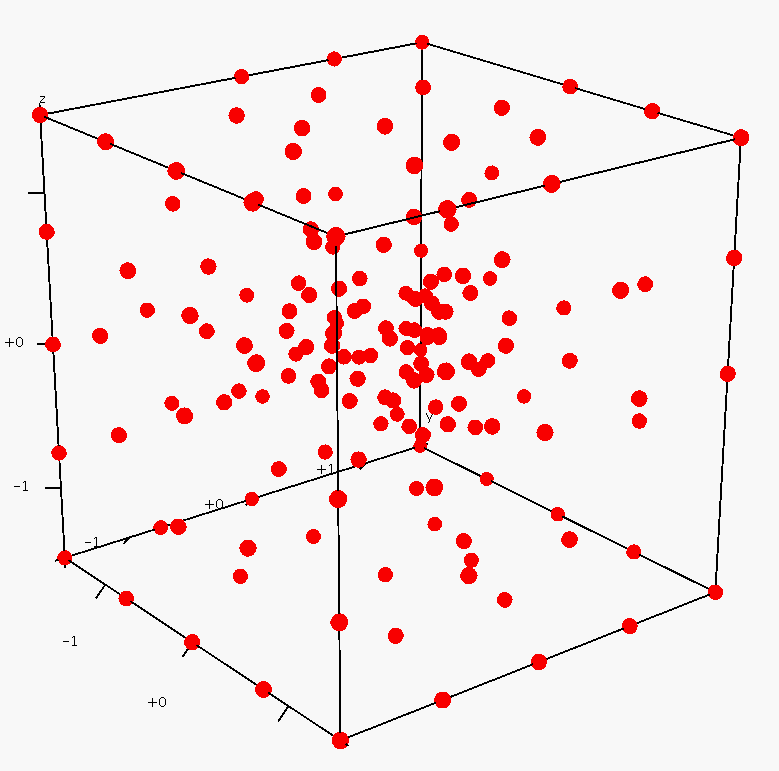


Figure 3.4: Tetrahedralization process. Closely related to Voronoi tessellation. [29][22]

#### Boolean operations

Boolean operations are also known as constructive solid geometry (CSG) and are used to perform volumetric operations between 3D geometries. For instance, two meshes can be added together to compute differences or intersections for mesh decomposition. Similar to a cookie cutter, it enables the decomposition of a mesh into smaller parts. This method is not automatic thus requires the work of an artist, making it a relatively slow technique. But on the other hand, it provides a high degree of control.

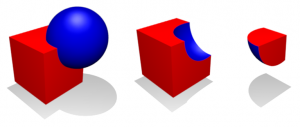


Figure 3.5: (Left) Sphere and cube are intersected. (Middle) The cube after calculating the difference.  
(Right) The remained piece. [55]

#### Convex decomposition

This method takes in a concave triangle mesh and divides it into smaller convex pieces. An artist can create a convex decomposition manually by using simple convex primitives such as boxes, spheres and capsules. It is possible of creating convex decomposition automatically, but it may involve tweaking of some parameters to get desirable results. Even though it is an NP-hard problem, an approximate method using top down or bottom up can still be implemented.

An example of a top down approach is a free implementation made by John Ratcliff. Concave meshes are recursively broken down into parts, until each part is convex.

An example of an implementation using a bottom up approach is Khaled Mammou´s Hierarchical Approximate Convex Decomposition (HACD), which was inspired by John’s implementation. This approach breaks source geometry into smaller parts in physically expected way, for example, the bunny’s ears come off, instead of breaking at some arbitrary position. The number and shape of those pieces however are determined entirely by the source geometry and does not provide any artistic control.



Figure 3.6: Convex decomposition can also be created manually by an artist, using convex primitives such as spheres, boxes and capsules.

### Real time destruction methods

Once the generation of pre-fractured assets has completed, runtime breaking methods are used for destroying objects during game execution. The objects are being fractured into separate meshes as part of the simulation and the physics engine also solves the object dynamics and collisions. Current available physics libraries include Havok Physics, Bullet, PhysX, and the Open Dynamics Engine.

#### Canned Animations

Game assets that are non-essential to gameplay and their simulation will not bring substantial value to the game are subjects for implementation through pre-determined animations. These animations are made in an animation tool and will play back when triggered or just play in a loop. Soft-body objects such as a waving flag or flowing water are major examples of this application. Another example is a distant skyscraper being tore down or demolished after a large explosion or earthquake. This approach is computationally cheap and saves a significant amount of processing power as the physically based animation is avoided altogether. The disadvantage of canned animations however is that the object will break the same exact way each time and not correspond to the interactions within the game.

#### Real-time Boolean operations

Boolean operations surpass the need of pre-made breakable objects and allow dynamic physical mesh damage to take place during runtime. This is achieved by subtracting a pre-modelled 3D shape from the original object on the exact point of impact. Because this is happening at run-time, the system dynamically re-triangulates the original model – by adding polygons as necessary - in order to maintain the original solid shape with the additional holes or missing pieces. The interior faces take the material and textures from the Boolean shape.

Prime example of this approach is the Geo-mod engine. Red Faction (2001) and Red faction 2 (2002) were created using the Geo-Mod (Geometry Modification Technology) game engine developed by Volition [6]. The engine featured the alteration of level geometry such as walls and ground by making holes in surfaces using certain weaponries. This was accomplished by creating a new “empty space” object at the point of collision. This new object would then be subtracted from the surface it just hit in real time – occluded by visual effects such as dust and explosion animations to hide the sudden subtraction of the meshes. As a result, modifying the geometry by adding a hole to it.

On one hand, the approach was unique at the time as it was not replacing an object to its damaged state rather than altering its geometry altogether. On the other hand, the feature arose some game design issues as the game could not protect the player from himself because they could tunnel through everything in the level thus making some areas “unwinnable” if enough damage was done to the surroundings. It made it more difficult for developers to set up various scenarios as it was harder to reason with the player’s unpredictable behaviour. Nevertheless, this was an important attempt to simulate a fully destructible environment in 3D regardless of the real-time constraints. Geo - mod 2 was later developed to create the next game in the Red Faction series, Red Faction: Guerrilla. This time the engine was focusing more on the destruction of buildings with a stress-based collapse model rather than the modification of level border geometry. A collection of smaller objects that are linked together with joints. Besides large or important structures, most buildings in Guerrilla are destructible and react to damage in real-time, eventually collapsing if enough structural support is lost.

#### Finite Element Method (FEM)

Finite Element Method as described by Parker and O’Brien [31] is a numerical technique for approximating complex solutions in engineering. It is widely used in the real-world analysis of determining what will happen to a structure when hit by another object or blown up.

FEM simulates the behaviour of a physical object which can be represented by a mesh made from smaller discrete pieces called finite elements. These elements contain material and structural qualities that determine the object’s reaction to certain physical circumstances. It enables the developer to define the mesh’s physical state by assigning different properties to each element to calculate its stress or temperature. Applications can be found in fluid dynamics; brittle fractures [33]; elasticity; heat transfer and other physical properties to simulate deformation, stress and fracturing.

FEM was folded in the digital molecular matter or DMM library. Its licence software was initially used for real-time destruction in video games. It was then advanced and upgraded for use in motion pictures through computer generated images [32].

#### Jointed breakable objects

Jointed breakable objects are pre-broken objects that use breakable constraints to hold the pieces together. The joints break when a certain amount of force such as a bullet is applied to it and the pieces start to fall by gravity. This method has a big impact on draw calls, memory consumption and physics calculations. It is also usually followed by a particle effect based on the surface type of the object. A clever way to work with it, is to use as few pieces as possible by spawning more pieces when combined with the particle effect.

### Terrain

There are two ways of generating terrain: One is to have a very fine mesh that can store sufficient information that allows the granularity of the resolution to be able to be replicated. The other is to dynamically generate it.

However, it is unlikely that large amount of information will be able to be stored in a way that allows it to be processed in an optimal way and that allows time for other processes to be executed as majority of processing time is being spent on rendering the terrain itself.

Some techniques can be applied to take a subset of the points and then interpolate between those points to produce features that can correspond to the sorts of ones that are expected to be found in terrains of the specific nature. That then offsets some of the needs to store information with the ability to be able to go through techniques that produce those dynamically as they are required. Hence, mathematical algorithms can be used for generating those as needed. That means that they do not necessarily have to store large amounts of data but store the routines and the production rules that enable to generate those features.

Terrain is important to a lot of games, as a model is very large so creating every point explicitly by hand is not feasible. What follows is an overview of some automated terrain generation techniques.

#### Heightmaps

A heightmap or height field is a 2D grayscale raster image used for storing terrain elevation data. A heightmap is a matrix, where each element represents the distance of displacement from the terrain grid. It is used to describe hills and valleys and as a grayscale image, black denotes the smallest height whereas white denotes the largest height and shades of grey signify in-between heights. Only the height of a specified vertex on the triangle grid is provided by the heightmap thus making it impossible to create overhangs, caves or tunnels. [43]

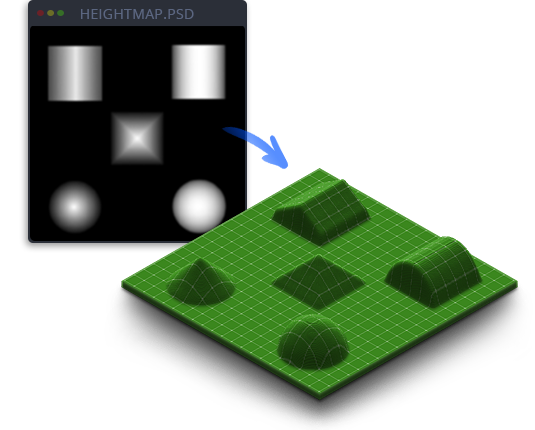


Figure 3.7: An example of a heightmap (greyscale image) and its use on flat grid. [42]

Usually a byte of memory is allocated for each heightmap element when stored on disk. Thus, the height value ranges from 0 to 255. When the range of values is not enough when matching the scale of a 3D world, it is possible to scale out of that range by allocating a float for each height element.

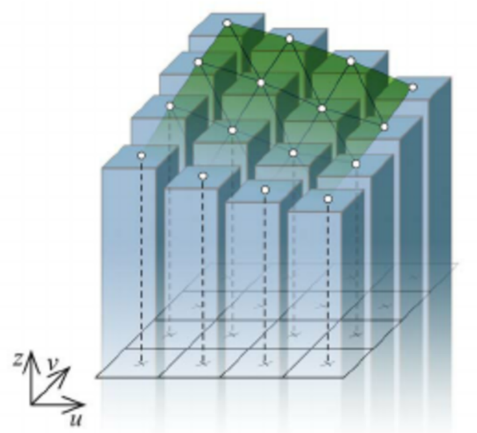


Figure 3.8: Representation of terrain as a heightmap from Holz et al [41]. In his paper, he analyses his implementation of height fields that can be updated dynamically.

#### Voxels

The word voxel derives from the combinations of the words “volume” and “element” and is the 3D equivalent of a picture element or pixel. Voxels are constructed from 3D regular hexahedrons or cubes that represent a single data point on a three-dimensional regular grid. A voxel is the smallest volume when dividing 3D space into discrete uniform regions [44]. Each volumetric pixel is defined by a position in 3D coordinates, the colour at that location or density as used medically for CT and RMI scan data images. This enables for the generation of complex terrain geometry that contains features not possible with heightmaps such as caves and overhangs [36].

An example of a game that utilised voxels to become a world-wide phenomenon is Minecraft [54]. Separating the vast procedurally generated game world into large blocks resulted in a fully destructible environment that was considered a desirable feature in a video game. Before Minecraft, destroyable environments were limited in the sense that some structures could be destroyed or damaged but not entire landscapes. A fully destructible world comes at a cost in terms of visual fidelity as everything is made of 1-meter cubes – even humans, creatures, animals as well as water and clouds. There is an algorithm however named “marching cubes” which can be used to “smooth out” the sharp corners of each voxel to give a more realistic terrain.

#### Marching cubes

The marching cubes algorithm was the result of the research that William E. Lorensen and Harvey E. Cline conducted for finding an efficient way of visualising data from CT and MRI scans for General Electric [37].

An implicit function in the form of f (x, y, z) = 0 takes in a point in space and gives out a value. The function then samples points in regular intervals inside a specified region. Some points represent empty space and some other points may be positioned inside or on the surface of some shape. The goal of the marching cubes algorithm is to construct the surface of that shape from triangles so it can be displayed as a mesh. In computer games, it is used to smooth out terrain generation made up of voxels by converting them to polygons before rendering them on screen.

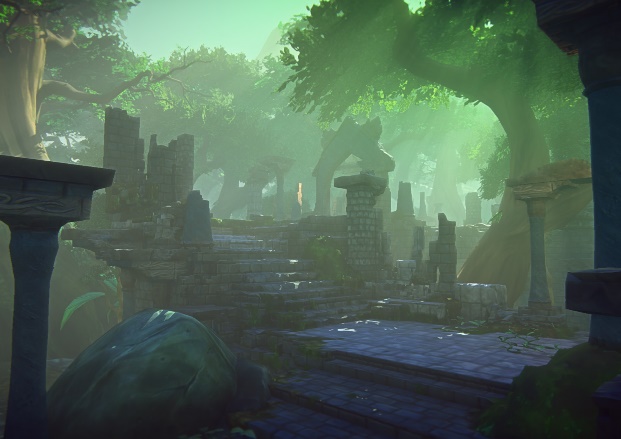


Figure 3.9: One game example that manage to combine voxels and marching cubes for the terrain and polygon meshes for the characters was Ever Quest Landmark.[40].

Ben Anderson [38] described it as follows:

Provided with an object, in order to test if an arbitrary point is within or on the surface of the object or bounds within the area which the object exists:

1. Division of the space within the region of interest into an arbitrary number of cubes.
2. Testing of the vertices of every cube for whether they are positioned inside the object. For each cube where some vertices are inside and some vertices are outside the object, the surface must pass through that voxel, intersecting the edges of the voxel in between vertices of opposite division.
3. Drawing a face inside each voxel connecting these intersections.
4. Source object is drawn.

To simplify the problem, we can break it down to a single cube. There are 8 vertices, and each can be inside and outside of the shape which results to 28 = 256 different configurations (see figure 1.8).

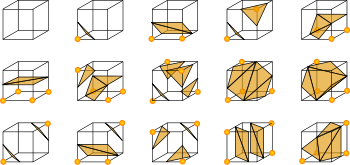


Figure 3.10: There are only 14 unique cases, the rest are just symmetries of those. [39]



1. (b)

Figure 3.11: For example, one point that is within the shape, a single triangle gets generated as seen in (a). If the point on its left is also inside the shape, a rectangle comprised of two triangles is also generated as seen in (b).

## Definition of the project specification

This chapter introduced several existing techniques found during research that are used for producing destructible environments. They were then categorised into breakable objects and deformable terrains. Taking into consideration the investigation conducted on destructible environments and the available time allocated for implementing a student project, one approach was selected to be employed and tested in a game environment. Being intrigued by the world of Minecraft [47] – its terrain in particular – and by comparing it with games such as Ever Quest Landmark [46] and Crysis 2 that used the marching cubes algorithm to “smooth out” the jagged edges of a terrain constructed from voxels, I saw the potential of having a fully or controlled destructible environment while also remaining visually appealing and keeping that desirable artistic control. The desirable objective is to generate a destructible terrain by producing a simple C++ game engine built upon a voxel-based design.

The generated terrain is going to be comprised of **voxels**,and **runtime Boolean operations** is the selected approach for deforming that terrain. Boolean operations need swift computational operations and the power of a graphics processing unit is enabled to be doing just that 60 times a second.

Similar to Minecraft, a photorealistic environment is not the aim of this project. However, several challenges arise with this approach, as voxel objects take O(n3) memory due to their cubic nature. In order to achieve efficient manipulation of voxels, solutions to problems involving effective use of cache memory, minimisation of polygon count and efficient communication between the CPU and the GPU need to be addressed.

The phased implementation is broken down into discrete development iterations. Each iteration will include a product back log subsequently prioritising the desirable elements using the MoSCoW method. Class diagrams will then be used to assist on directing the program’s structure. At the end of each iteration there will be a test plan, test result and iteration conclusion.

MoSCoW is a prioritisation method usually found in software development for better understanding and prioritising the development of key features that are expected to be included in the software [48]. The letters stand for:

* **M**ust have: Key Features that cannot deliver a viable solution without them.
* **S**hould have: Features that are important but not vital.
* **C**ould have: Features that are desired but less important
* **W**on’t have: Requirements that have been considered but decided not to be included in solution.

In conjunction with the product backlog, at the start of each iteration, there will be a prioritised brake down of tasks that should be implemented. After each iteration, a growing class diagram will provide an overview of the system.

Class diagrams describe the structure of a system and the relationship between each class. A class notation will provide information about a class’s name, attributes and operations. Lines that connect different classes together will illustrate the different types of relationship that each class have. These types may include simple association, inheritance, aggregation or composition and dependency [49]. After a feature or method has been implemented, a test plan and test result will be used for keeping a measurable element to be compared against at the end of each iteration.

A test plan defines the scope and methods which are going to perform during feature testing [50], consequently outputting a result that is going to be compared against when measured at the end of each iteration. The test plan for this project is going to include functional tests for testing if the software is operational, but also benchmark performance tests that are going to be used for tracking any performance issues of particular methods that were implemented during development.

The test plan is going to outline what the performance tests are such as the frame rate, memory usage and possible cache misses. Starting off with tracking the initial state of the framework when no feature has been implemented and then comparing against that for all the iterations. After implementing a function that slows the system down considerably then the test results will be compared against the benchmark tests and what should the function be expected to do. Thus, allowing for some additional reflection and possible issues that will need to be addressed and further tested. The tests are conducted using Visual Studio’s diagnostic tools where memory and CPU usage are provided from profiling tools at runtime. Frames per second are recorded by ImGui built-in functionality.

For the list of specifications of the hardware in which the artefact was tested on, see appendix A.

The following is an outline of the planned iterations with brief descriptions of main objectives:

## Outline of proposed iterations

**Iteration 1** – Program structure and design:

In this iteration, a suitable application programmable interface (API) is going to be chosen, a product backlog is going to be generated and the first draft of the program’s architecture using class diagrams is going to be produced.

**Iteration 2** – Framework set up:

In this iteration, the provided framework is refactored to assist on separation of Direct3D initialisation and the actual implementation. A “free – look” camera movement will enable the user to navigate around the scene with ease.

**Iteration 3** – ImGui integration and Voxel class:

In this iteration, Immediate Mode Graphical User interface (ImGui) will be integrated into the framework to assist on development. ImGui is an open-source tool, licenced under MIT licence. It is a graphical user interface used for simplicity and productivity by outputting optimized vertex buffers that can be rendered anytime in a 3D-pipeline enabled application [45]. This will allow for instant and organised data access to relevant information of an object but also immediate update of any transformations, adjustments or flags. After ImGui integration, a definition of a voxel class and rendering of a voxel grid will be the next logical step.

**Iteration 4** – Terrain generation:

Once the basic framework has come into place, the terrain generation using voxels is going to be developed.

**Iteration 5** – Boolean operations:

Once the terrain of voxels can be generated, adding destructibility to the environments using brushes and Boolean operations will be the next step.

**Iteration 6** – Testing and optimisation:

Once the destructible environment has been produced, time for further tests and optimisations will be given.

# 

## Iteration 1 – Program structure and design

In this initial iteration, an API is selected, and the program’s architecture is introduced and demonstrated using a generic class diagram. Followed by a breakdown of the prioritised desired elements using the MoSCoW technique.

### Selection of Application Programmable Interface (API)

There are many APIs available but due to author’s past experience the one that is going to be used for this project is DirectX, version 11. According to Frank D. Luna, Direct 3D is a low – level rendering library used for writing high performance 3D graphics applications. Its API closely models the underlying graphics hardware it controls [43]. The framework is provided by one of the course’s modules named “Advanced graphics and real-time rendering”. This framework is supplying the building blocks for initialisation of Direct3D, texture loading, shading, basic camera set up and the rendering of a cube. The framework is then refactored and built upon to fit the needs of the project.

The chosen programming language for developing the engine is C++.   
C++ is compiled straight to machine code, unlike other languages that use a virtual machine with an abstraction between programming logic and assembly logic. Examples of these languages are Java which uses the java virtual machine (JVM) [57] and C# where .NET uses the virtual machine of Microsoft called Common Language Runtime (CLR) [56] and Mono uses the Common Language Infrastructure (CLI) from ECMA international [58], thus nominating C++ a viable candidate for low level programming that efficiently and directly manipulates resources. It can also be used as a high-level object-oriented language for taking advantage of encapsulation, polymorphism, templates and abstraction. Additionally, One of C++ important design principles is that programmers do not have to pay for features that are not being used. For instance, a virtual table are only generated when there is at least one virtual function in a class. [51]

### Product backlog, MoSCoW

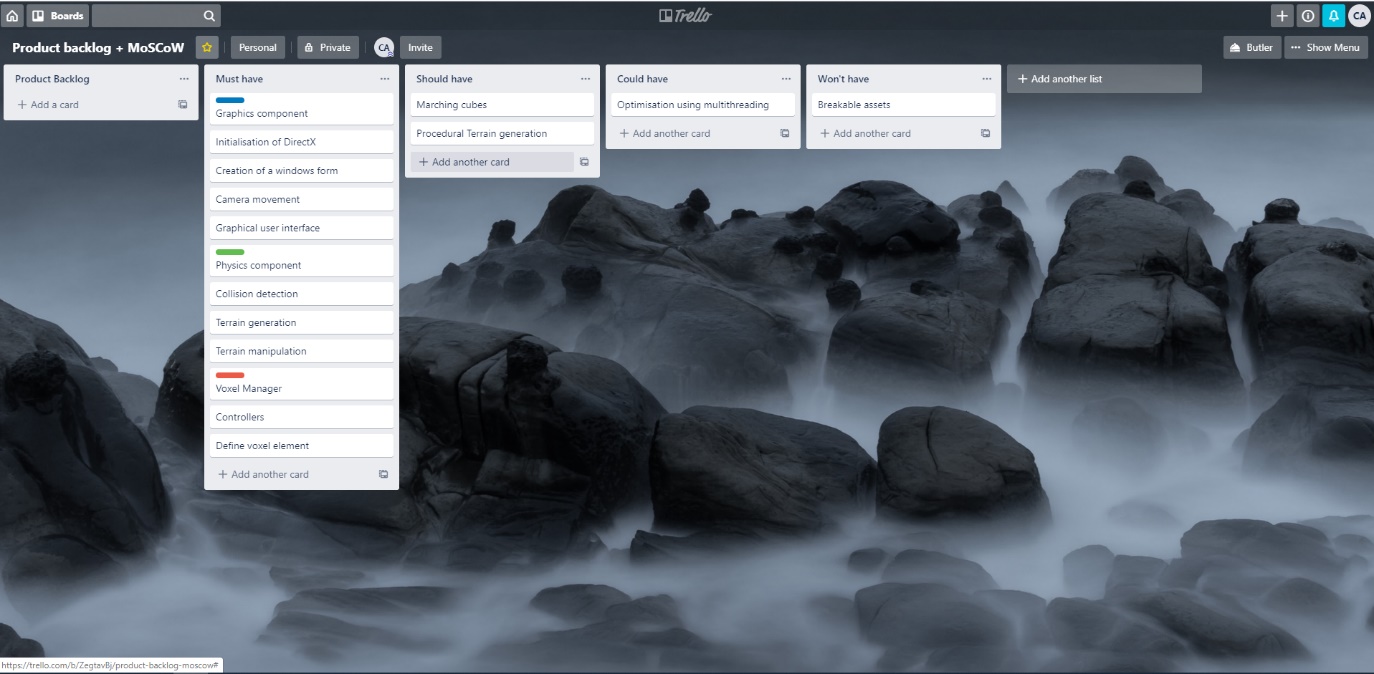


Figure 4.1: This example demonstrates a product backlog being broken down into prioritised tasks using the MoSCoW technique.

**Must have**

In order to build a 3D voxel-based engine, a graphics component for rendering the terrain on screen is essential. For making the environment destructible, a physics component for handling collision detection between the voxels and the brush used for Boolean operations is a requirement. A graphical user interface for manipulating voxels would greatly benefit the development of the environment.

For the first iteration, there should be a workable camera set up and the initialisation of Direct 3D should be refactored into its own class to separate from the application’s functionality.

**Should have**

Efficient implementation is the main goal of this thesis thus attempting to optimise the terrain generation and manipulation by separating the tasks into different threads. Another desirable element is procedural terrain generation to avoid manually generating terrain which is time-consuming but also non-efficient.

**Could have**

Further expansion of the voxel engine is to “smooth out” the surfaces using the marching cubes algorithm. Due to the complexity of the algorithm and time constraints this expansion is desirable but not essential.

**Won’t have**

Based on selection of approach, the generation and manipulation of terrain using voxels is the purpose of this project thus excluding other approaches that focus more on destructible 3D mesh models.

### Class diagram

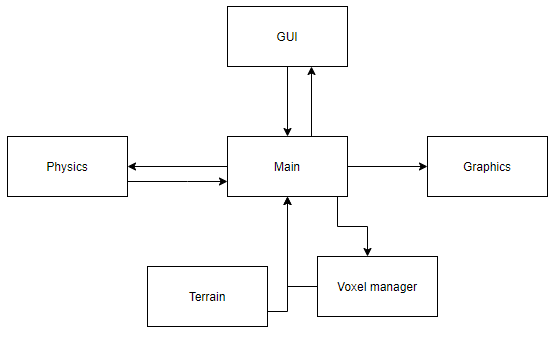


Figure 4.2: Diagram displaying architecture of the application.

### Test objectives

The test objectives for the first iteration are to establish the benchmark performance values that are going to be compared against in later iterations when new features have been added to the framework. One of the variables that will be taken into consideration is the number of frames per second which will have the highest impact in user experience thus indicating the overall performance of the application. Achieving the highest possible frame rate is important but not essential. Next, information provided by diagnostic tools in visual studio such as process memory and percentage of CPU usage is going to be recorded through each iteration to ensure that any bottlenecks between the CPU, GPU and RAM are revealed. For the test results, see iteration 1, Appendix B.

|  |
| --- |
| Test cases |
| Frames per second with a single object being drawn on screen |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |

Table 4.1: Benchmark test cases when a single cube is rendered on screen.

### Review

The first iteration was about selecting an API, producing a product backlog and prioritising the desirable features using the MoSCoW method. In addition to these tasks, further time was spent on studying the provided framework and ensuring that the initialisation of Direct3D and basic camera set up were in place. As you can see on the Gantt chart above, these tasks were set in parallel with the last part of research analysis and I allocated more tasks that I could achieve thus pushing some further functionality such as camera movement and defining a voxel to the following iterations.

### Retrospective

On reflection, I gave too much time to myself for certain non-programming tasks but on the contrary when this was realised, I gave much more tasks to complete until the end of the iteration. I will take that into consideration and allocating tasks during the next sprint plan.

## ITERATION 2 – framework set up

In this iteration, the framework provided by “Advanced graphics and real-time rendering” module is refactored. The initialisation of a window, initialisation of Direct3D, basic camera set up, lighting and rendering of a textured cube are all taking place inside the main function.

### Product backlog, MoSCoW

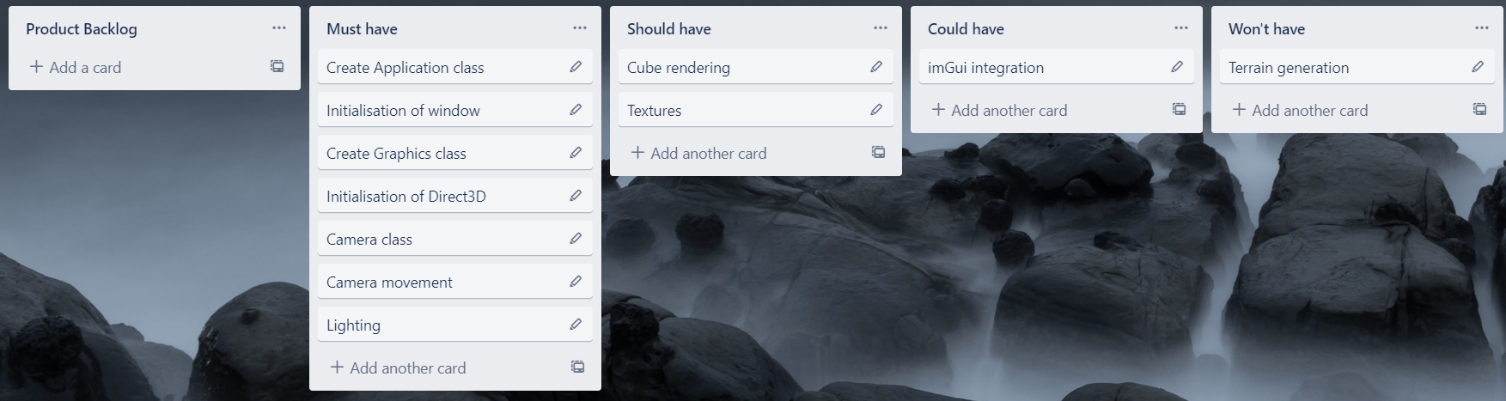


Figure 4.3: The aim of the second iteration is to get familiar with the framework and refactor it so that it can be easily expanded to fit the requirements of the application.

**Must have**

The class in which the main function exists should only contain the main function. All the rest of the functionality including the initialisation of Direct3D, initialisation of application window, initialisation of 3D scene and rendering of the camera and the cube should be implemented separately into different classes. The camera class should encapsulate the view and projection matrices and update them in its own class called “Camera” while also implementing relevant attributes that allow for updating the camera’s position and rotation. The Graphics class will be responsible for initialising Direct3D and imGui related functionality. The Application class is where the initialisation of a window and all the updating and drawing of our application is going to be called. Hence, the graphics, camera and game object are all being called inside the application class.

**Should have**

Once the refactoring has finished, a textured cube should be rendered at the origin of the scene by the end of the iteration. This will provide the building block of our environment.

**Could have**

ImGui would benefit development greatly because is a tool that makes updating information at runtime much easier. It provides a panel in which things such as transformation, textures, overview of an object’s attributes can be viewed and updated at runtime resulting to quicker tests and debugging.

**Won’t have**

The step of refactoring and studying the framework to fit the needs of our implementation proceeds before the terrain generation starts.

### Class diagram

****

Figure 4.4: A class UML diagram showing the current structure of the implementation.

**Graphics**: Initialises Direct3D

**DirectInput**: Receives direct input from mouse and keyboard using IDirectInputDevice8 object.

**Application**: Main class where it controls the application procedure.

### Test objectives

The test objectives in iteration two are to ensure that the refactoring that is going to be made in the framework is not going to have any major impact on the performance of the application thus similar results to iteration 1 are expected. Additionally, testing that the camera is functional and the ImGui library was imported correctly by rendering some examples on screen. For the test result, see iteration 2, Appendix B.

| Test cases |
| --- |
| Frames per second with a single object being drawn on screen |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Does application class initialise window and calls graphics class? |
| Does Graphics class initialise Direct3D? |
| Is the Camera class functional by creating a camera object? |
| Is a voxel being rendered? |
| Can the camera move with key inputs? |
| Can the camera move with mouse input? |
| Is ImGui imported in the project correctly? |
| Is ImGui initialised? |
| Does ImGui receive mouse input? |

Table 4.2: Tests demonstrating the issues occurred during the second iteration. These tests include the encapsulation of the initialisation of Direct3D, the import of ImGui into the project, creation of a camera with mouse and keyboard input for a “free-look” movement.

### Review

In the second iteration, I finished refactoring the framework. I separated Direct3D related functionality into its own class called “Graphics” and everything else that relates to the actual implementation in the “Application” class. A recap on initialisation of direct3D was needed for debugging issues occurred during refactoring. A camera was created and separated into its own class to compute all the needed matrices in one place. In Addition, a directed input class replaced Win32API to get input from the user. Win32API was designed for applications where users enter data through keyboard and not for application where quick real-time input would be ideal. There is a lot of extra processing when getting input from Win32API as it needs to convert each key to ascii while also there are some extra steps for windows special keys such as “alt” which is redundant to games as a frame cannot be rendered until the message queue has emptied. As a result, a “free-look” camera movement was implemented by receiving direct input from mouse and keyboard.

### Retrospective

Up to this point, the sprint goals have been mostly met as the API has been chosen and the basic framework of the application has been refactored. It could have been quicker if knowledge of initialisation of direct3D was thoroughly understood from the start as the debugging process would speed up. Better time management should be taken into practice as more time is consumed on writing and coding and less on planning ahead which if done too late, I can be easily caught up with irrelevant tasks and lost track of my goals. On the other hand, having a framework that I understand deeply helps on debugging issues that may occur in the future in regards of voxel rendering.

## ITERATION 3 – Voxel placement

In the third iteration, an attempt of placing voxels manually in the environment with mouse input is taking place using two approaches:

One way to generate a voxel is by rendering and clipping it next to the default block that was first generated at the world origin. This can be done by casting a ray from the camera and onto the surface of the cube. Hence, as the size of the cube is known, using an offset towards the camera from the selected cube, a new voxel can be generated and placed right next to the original one.

The second approach is to generate a block made from smaller voxels by dragging the mouse across the screen. The idea is to store the last and new position of the mouse and get the cross product of the generated vectors consequently creating the orthogonal vectors that form the other side of the block from the dragging of the mouse button. Thus, when the mouse button has been released, it outputs the size of the block.

In order to select a voxel with the mouse, it is essential to transform the 2D position of the mouse cursor in screen space into a 3D ray in world space that pierces through the voxel we are interested in and checks if and on which point on the ray it intersects with it. The implementation of “picking” was taken by Braynzar soft Direct3D11 tutorials [52].

After implementing picking and testing that all triangles on all sides of the voxel intersect with the 3D ray, the next step was to create multiple instances of a voxel. Thus, hardware instancing was the next logical step that came to mind. Hardware instancing is a technique of rendering multiple copies of the same mesh with slightly varied attributes and transforms. As this iteration was coming to an end however and instancing was not yet successfully added to the framework, I decided to scrap the idea as I discovered the concept of chunks that I am describing in the next iteration.

### Product backlog, MoSCoW

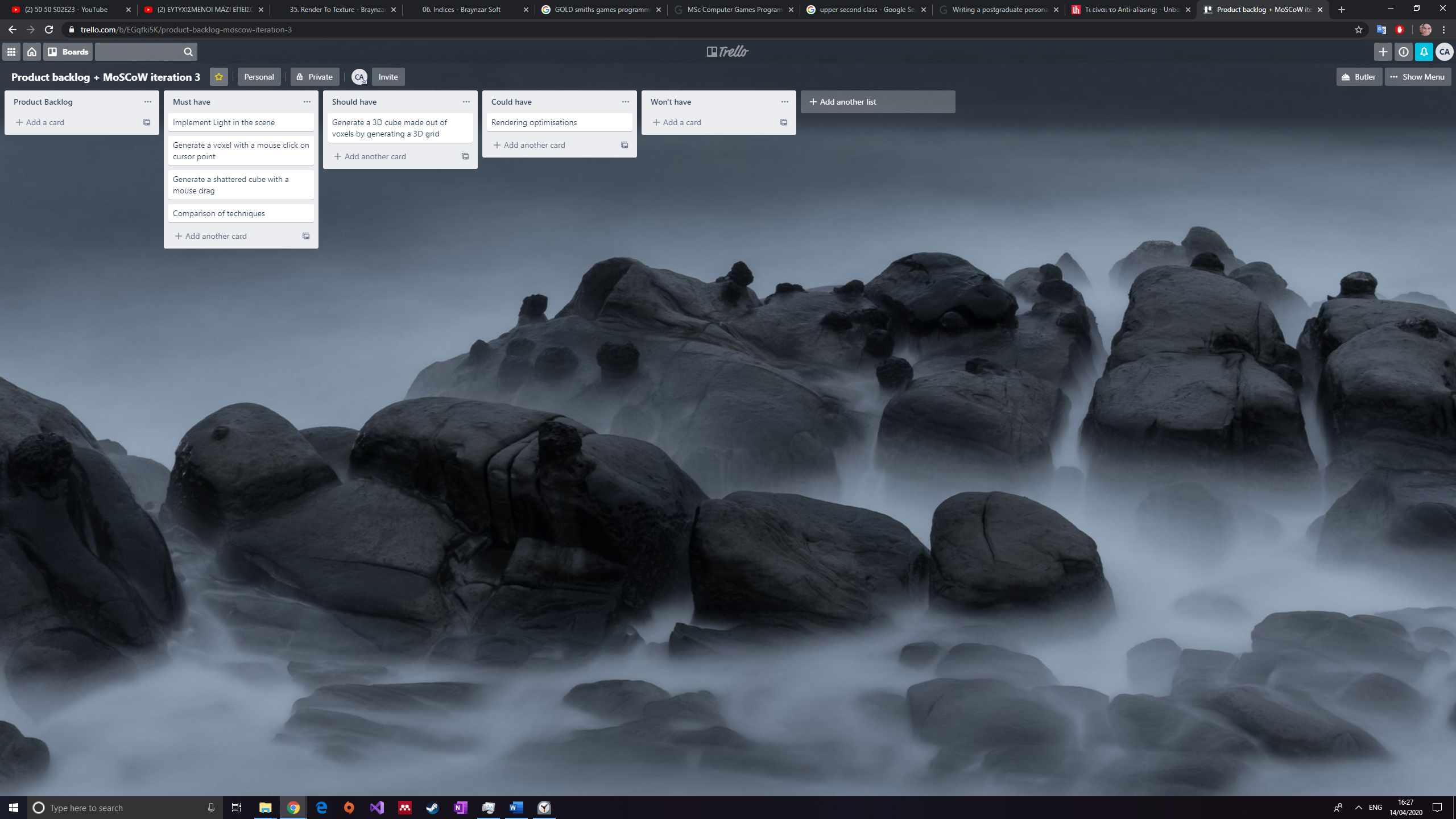


Figure 4.5: The product backlog for the third iteration includes the basic implementation of a light in the scene and the generation of voxels with user input.

**Must have**

* Basic lighting setup must be implemented to illuminate the scene. This can either be the enablement of a point light or a directional light that were included in the framework beforehand.
* Placing voxels next to each other by generating a cube mesh with mouse input at the point of the cursor.
* Generating a bigger block that is comprised of smaller voxels by dragging the mouse across the screen.

**Should have**

Implementing the two approaches should be the two high priorities for this iteration. If there is still time after completion and testing of these approaches, another idea is to generate 3D grid of voxels that make up a bigger block using a nested for loop.

**Could have**

Rendering each voxel separately is not efficient as not all the sides of a voxel will be visible to the camera in a single frame but also voxels that occluded by other voxels are extra burden when rendering the scene. Finding a better way to optimise the rendering of voxels by rendering only the sides that are visible to the camera.

**Won’t have**

Experimental stages exceed the generation of a larger terrain.

### Class diagram

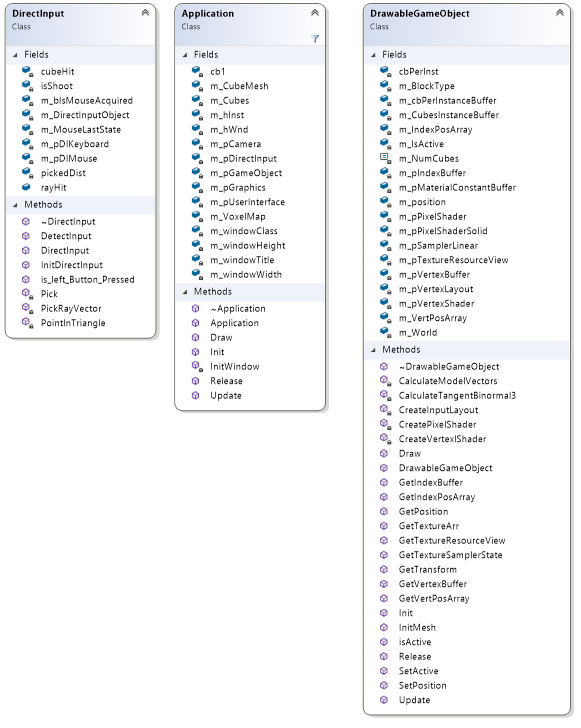


Figure 4.6: A class diagram displaying the added functionality.

**DrawableGameObject**: it was built in with the initial framework. Initialises and renders a primitive mesh, in this case is a cube.

**DirectInput**: Pick() was added to the class to receive the position of the mouse in screen space. Transform the mouse position in a 3D ray in world space and iterate through each triangle to check if the ray has intersected with the cube.

**Application**: Both direct input and DrawableGameObject objects are called inside the Application class.

### Test objectives

The test cases of the third iteration make sure that ray casting/picking was implemented successfully. For the test results, see iteration 3, appendix B.

| Test cases |
| --- |
| Frames per second with a single object being drawn on screen |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Is clientWidth receiving the screen’s width from WndProcc |
| Is clientHeight receiving the screen’s width from WndProcc |
| Is PickRayVector() receiving data from clientWidth |
| Is PickRayVector() receiving data from clientHeight |
| Value of t (ensuring that we do not pick objects behind the camera) |

Table 4.3: Testς cases during the implementation of ray casting.

### Review

The third iteration included the addition of ImGui, implementation of ray casting, manually placing voxels in the game world and generate an array of voxels with mouse drag. Only some of the objectives were achieved. Resulting to reflection and a new set of objectives. For tracking progress of the project, I am using a log to hold any thoughts and notes that I might look at the next day, a Trello board for everyday and iteration tasks, and lastly TeamGantt to keep track of my objectives.

### Retrospective

In the third iteration, I started working on how I am going to create a world made of voxels. I first imported ImGui into the framework to support development and make an editor to help with debugging. After that, that first feature to implement was to make each voxel to be manually placed using the mouse. I didn’t know about ray casting in DirectX so researching and making that work at the end of was what took most of my time for this iteration. I managed to add ray casting to the framework, but I didn’t manage to achieve the goals that I had set for myself at the start of the iteration. Although that slowed down development, I stumbled upon the concept of chunks and realised that manually placing each voxel in the world will mean extra work for the graphics card for each frame even with a significantly small number of voxels. As a result, I changed my approach for the next iteration.

## ITERATION 4 – Voxel organisation

In this iteration, the concept of a chunk is introduced and included in the framework to group and render a multi-dimensional array of voxels more efficiently.

A chunk refers to the arrangement and organisation of voxel space. In order to generate a large and expansive world that the engine would work with infinitely in all three planes then the use of chunks is what would minimise the number of draw calls at each frame but still provide a satisfying number of rendered voxels. To create a chunk, I started with rendering multiple voxels next to each other using a fixed multi-dimensional array. That demonstrated what a single chunk would look like but also how processing-heavy it can be to the graphics processing unit. A voxel comprises of six faces with two triangles each for a total of twelve triangles. Hence, generating a 256x256x256 world of voxels would result to surrounded voxels that are placed at the most outer layer of a chunk. ‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬surrounded voxels that are placed at the most outer layer of a chunk. ‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬surrounded voxels that are placed at the most outer layer of a chunk. ‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬surrounded voxels that are placed at the most outer layer of a chunk. ‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬

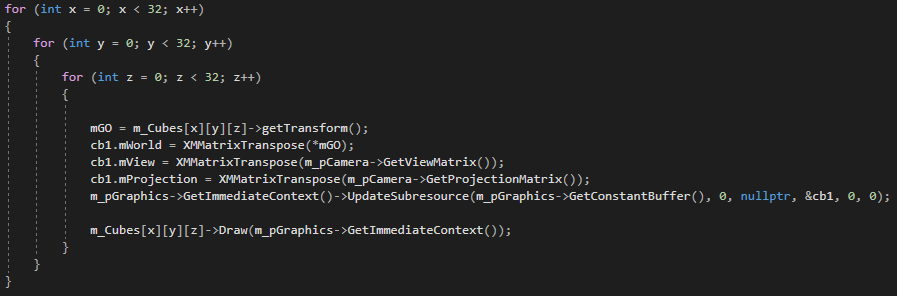


Figure 4.7: A multi-dimensional array of voxels being rendered. In this example, the draw function is called for each voxel.

The advantage of a chunk is that it makes a batch of voxels that only requires a single draw call for each chunk. Therefore, significantly reducing the rendering overhead. Assuming that a chunk represents 16x16x16 voxels then a 256x256x256 world of voxels would only need to draw 4096 chunks instead of 16.8 million voxels.

There is a pitfall however when it comes to using chunks. As this project aims at modifying the capacity of a chunk by disabling or enabling the voxels it contains, it will be necessary to rebuild the chunk render data each time it alters. In a case where a voxel lies on the border between two chunks, the chunks will need to be rebuilt [36]. Therefore, finding a suitable number of voxels in a chunk will depend on performance tests as having a large chunk will result in less rendering overhead but also in slower reconstruction times.

### Product backlog, MoSCoW

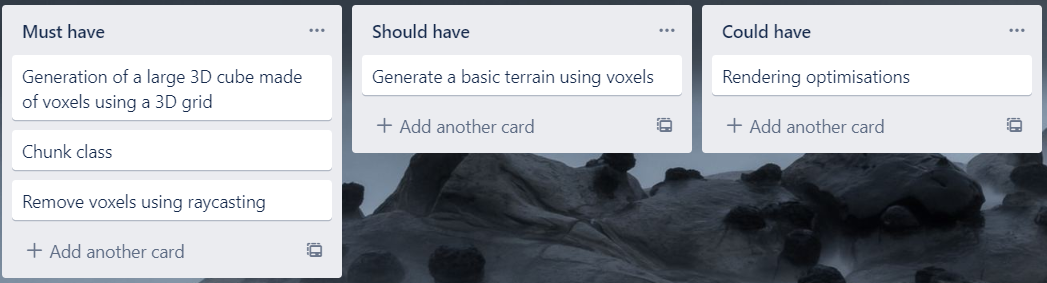


Figure 4.8: The product backlog of the fourth iteration. In the iteration a chunk prototype is generated, and the chunk class is used for initialising and rendering chunks.

**Must have**

* Generation and definition of a chunk; the building block of the generated terrain.
* Generation of a 3D array of voxels that make up a bigger cube. This will give a glance of how a chunk will look like.
* Addition and removal of voxels using ray casting. Basic implementation of a destructible terrain.

**Should have**

* After the generation of a single chunk, a basic terrain can be produced by separating empty voxels to filled ones. This means that for each “air” type voxel, the chunk generator will skip until it finds an “earth” type voxel.

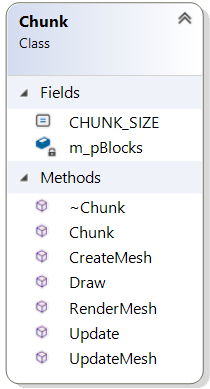
**Could have**

* By generating a chunk, there are going to be many voxels that occluded by voxels stationed at the outer layer of the chunk thus the renderer does not need to render all the sides of a voxel unless there no “solid” voxel next to it. This will improve rendering performance making it more efficient to render more chunks that will result in a larger terrain.

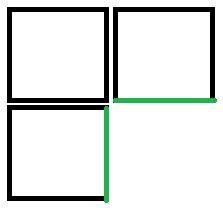
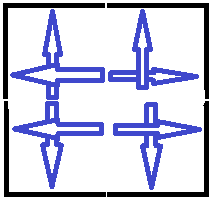
**Won’t have**

* Any algorithm that causes turbulence or noise will not be implemented in this iteration.

### Class diagram

**Chunk**: the chunk class is responsible for the generation and management of a single chunk. Starting in the constructor where a 3D array of voxels is initialised. It is worth noting that it is not essential to store voxel data in a 3D array but makes it easier to read if, for example, a voxel is placed at 5,8,3 then it can be accessed in the array by calling m\_pBlocks [5][8][3]. By adding a flag to each voxel to determine if it is active or not, each voxel can be rendered or skipped based on that flag.

The **CreateMesh()** function is where each cube mesh is initialised and placed next to each other to make up a whole chunk. The way that this works is we are iterating through each block position in the chunk and translating a cube to that position. As each block is a drawableGameObject, it means that a vertex and index buffer as well as a vertex and pixel shader are created for each cube. Consequently, invoking a draw call for each individual voxel. An optimised version of this implementation that would have gained significant performance boost from the fundamental concept of chunks would be to first define the size of a chunk and then merging all voxels into a single mesh by creating one vertex and one index buffer for the whole chunk. Therefore, making a single draw call for each chunk instead of making a draw call for each voxel in the chunk. Additionally, all the voxels being rendered are inside the chunk meaning the user would not see them anyway. With the current implementation, an approach I took to solve the rendered triangles inside the chunk is, given a chunk, I look at every single block in it and then for each block I look at its neighbouring blocks. First, if a block is surrounded by activated blocks, then I set another flag stating that it is an empty block thus it should not be rendered. If the neighbouring block is deactivated, then I include the indices for that block face, as seen in Figure 4.9. In the case that a neighbouring block is activated, then I include the indices for the opposite block face, also seen in Figure 4.9.

1. (b)

Figure 4.9: (a) Include face if neighbour block is deactivated (b) Include opposite face if neighbour is activated.

Although this approach was a step towards the right direction as the number of rendered triangles decreased (see table 4.5), each voxel in the chunk is still being drawn separately.

### Test objectives

The objective of the tests in the fourth iteration are for testing the performance of chunk rendering in different chunk sizes. Additional tests are for recording the performance after the optimisation of occluded triangles was applied. For the test results, see iteration 4, appendix B.

| Test cases |
| --- |
| Frames per second with a multi-dimensional array of 2x2x2 voxels |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Frames per second with a multi-dimensional array of 4x4x4 voxels |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Frames per second with a multi-dimensional array of 8x8x8 voxels |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Frames per second with a multi-dimensional array of 10x10x10 voxels |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |
| Culling of faces that are occluded by adjacent voxels |
| Skipping occluded triangles |
| Empty blocks |

Table 4.4: Comparison of performance test between different chunk sizes. Additionally, testing of an optimisation of occluded triangles.

| Test cases |
| --- |
| Frames per second with a multi-dimensional array of 10x10x10 voxels |
| Process memory at runtime (in megabytes) |
| CPU usage (% of all processors) |

**Table 4.5: Test cases after applied solution.**

### Review

According to the initial plan, the goal of the fourth iteration was to start generating a terrain in which I could then use methods such as Boolean operations to modify the terrain at runtime. This objective was not achieved by the end of iteration 4. However, a definition and rendering of a chunk which is going to be the basic building block was completed with some attempts of optimisation. There is still another major optimisation to be made so the rendering of a chunk can be much faster and less graphics heavy.

### Retrospective

What I managed to accomplish in this iteration is the introduction of a 3D array of voxels that are grouped together to make a chunk. A chunk is going to play an important role in terrain generation and be the building block of our terrain. I was not satisfied with how each voxel was being rendered separately with all the voxels inside the chunk also being rendered. Although I managed by the end of the iteration to skip rendering occluded triangles, a draw call is still being used for each individual voxel. Hence, memory used has been decreased but the number of frames per second remain the same. The sprint objective was to generate a terrain procedurally using voxels. Even though the goal was not met, an approach was taken towards this goal by first generating a chunk. I also tried to implement a debug log using ImGui but as I couldn’t import an example into the project, I found a different example that used the visual studio’s output window.

# 

This chapter evaluates the success of the project, any improvements that could be done to the project if it were to be repeated as well as any experience gained throughout development.

## Evaluating the success of the project

### experience gained

After determining what the artefact was going to include, my biggest challenge was to relearn the DirectX 11 API so I could use this knowledge to refactor and debug the framework provided from one of the optional modules. By giving myself a few weeks to get back on track, I followed BraynSoftTutorials [52] that helped me understand how to initialise a Window and Direct3D to render a shape in 2D and a cube in 3D. This gave me enough support for continuing and working on the project.

Taking advantage of profiling tools available in Visual Studio such as diagnostic tools, particularly CPU and memory profiling gave me internal look at where most memory resources and CPU processes are utilised. For instance, generating a 10x10x10 chunk without any optimisation meant rendering 12.000 triangles per frame but using 2.2 Gigabytes of memory at runtime resulted in requiring 18.3 milliseconds per frame. This indicated that it was a GPU bottle neck due the plethora of voxels needed to be rendered. In conclusion, profiling tools can reveal bottlenecks and record the performance of the application.

Learning the importance of project management and utilising different project management tools supported an agile type of development and helped in tracking progress and setting objectives. For example, I kept a logbook that I updated daily. It benefited me in keeping notes, reminders or questions that I would look at the following day or sometime in the future. Trello boards were used to keep track of everyday tasks by separating each one in TODO, INPROGESS and DONE cards. In addition, I created another board to produce the product backlog and allocate each task using the MOSCOW technique. The product backlog included important features and should be added to the following iteration. Another tool I utilised was a Gant chart. At first, I used a free sample of a spreadsheet file. The spreadsheet was inflexible and offline which meant valuable time was spent organising and backing up the Gantt chart than working on the project. However, the user interface of TeamGantt proved to be easier to handle; the tool is free; online and can be integrated in a Trello board. TeamGantt was used to make my Gantt chart automatically and assign dates to each task I had planned for each iteration. Making milestones and colour coded bars assisted visualise how much time was allocated for each task. To summarise, implementing an agile development approach meant using various project management tools that kept track of the project’s progress and objectives for each iteration.

### final conclusion

The environment in the real-world is frequently modified due to natural phenomena or human intervention. The desire of transitioning from traditionally generated static virtual scenes to a more dynamic and interactive virtual environment is becoming ever so strong over the last decade. Increased obtainability of computing power; multi-core processors and advances in computer graphics research and development have made it possible to simulate destructible environments at runtime.

This thesis has investigated popular methods employed to simulate destroyable environments, emphasising more on object destruction and terrain deformation. Additionally, one technique was chosen to be implemented and tested in a simple 3D environment using the DirectX 11 API.

Before any research was conducted, a project planning methodology was chosen to manage the project. An agile approach was selected with some slight modifications to fit a solo project. The approach involved a variety of tools to support development but also to hold tasks and objectives that were broken down into several iterations.

Alex Andrew’s “Srum of One” [17] version of agile was chosen to be followed but it was not followed to the letter. For example, a product backlog was used to list all the individual features that were thought to be important enough to be implemented on the following sprint. However, the method of numbering the tasks depending on the difficulty of each one to keep track of productivity levels was tested during research but was not used properly as the tool used for tracking the numbers – Microsoft’s OneNote - was not used frequently thus it was quickly forgotten and later abandoned. Another example is daily scrum. A text file was used initially to answer questions such as what was accomplished the previous day, what was going to be done the same day and what obstacles blocked uncompleted tasks. A personal logbook proved to more useful though. The logbook could be accessed online and was used for making notes, updating what was accomplished in the day in regards of the project and for leaving any questions that would require further research. Trello proved to be a very useful online tool to make the product backlog, MOSCOW, track daily tasks as well as use it concurrently with TeamGantt to set tasks and deadlines for each iteration that would be represented automatically in a Gantt chart. Story time was merged at the end of each iteration with review and retrospective to brainstorm new objectives to add to the next iteration. However, review and retrospective for the later iterations were skipped and became outdated which meant that I had to refer to the logbook to remember what happened through that iteration. A better implementation of this approach would be to be more organised by focusing on staying on the timetable so that for each day is clear what the main tasks are and for the daily tasks I should have included in the logbook what was blocking me from finishing a certain objective so that I would not get side-tracked and spending time on lower priority issues that were sometimes irrelevant.

After the research was completed, deciding what the artefact implementation was going to include was something that pushed back development as I couldn’t make my mind up if the artefact was going to be about object destruction or terrain deformation. What helped me decide was Atomontage. Atomontage [59] is a volumetric graphics and simulation software that its main purpose is to replace polygons. Graphics accelerators have been designed to operate with triangles in exceptionally high speeds. However, vector-based polygon meshes are represented by triangles which can only model surfaces, thus creating an insufficient approximation of the real world. Voxels on the other hand, are volumetric elements in a 3D grid meaning that everything in the scene is made of voxels only the “air” type voxels are not rendered. According to Atomontage, voxels can represent the full depth of matter consequently simulating a more accurate model of reality. I wanted to investigate the nature of voxels more deeply and understand the technical requirements needed to generate interactive terrain in a suitable game scenario.

The two main objectives of the artefact were to generate terrain using voxels and modifying it using runtime Boolean operations. Further implementation would involve the marching cubes algorithm to smooth out the terrain and improve the visual fidelity of the scene. The objectives were partially met as more emphasis was put on the generation and optimisation of chunks that would later prove to be the building block of any vast procedurally generated terrain made of voxels.

As you can see in the Gantt chart in iteration 1, appendix C, the goal of the sprint was to get familiar with the initial framework and create a separate camera class that would allow the user to move freely around the 3D environment. It was later decided that refactoring of the framework was required as both initialisation of Direct3D and rendering of a mesh was taking place inside a single class. Therefore, by the end of the iteration, it was expected to have everything set up and start defining a voxel class but refactoring of the framework preceded thus the objectives were modified for the following iteration.

Moving on to the second iteration, some of the objectives were met and some were partially met. The initial goal was to create a camera that would allow the user to move freely around the 3D environment. DirectX11 API was chosen due to author’s past experience. However, a number of things had to be relearned in order develop and debug using the Microsoft’s API. The framework provided was all implemented into one class thus it had to be refactored in order to be more manageable but also to detach any Direct3D initialisation code from the actual application. This slowed down development as time was spent researching and following tutorials that would enable this refactoring to take place. Therefore, by the end of the iteration, it was expected to have everything set up and start defining a voxel class, but it was not met due to refactoring of DirectX framework (see iteration 2, appendix C).

In the third iteration (see iteration 3, appendix C), not all the objectives were met due to technical errors and insufficient experience relating to instancing in DirectX. The sprint objectives were about importing ImGui library and using two different techniques to generate voxels manually in the 3D environment. One technique was to use ray casting for generating a voxel depending on the position of the mouse cursor and the second technique also involved ray casting, but this time generating an array of voxels by dragging the mouse across the screen. This introduced two problems. The first one was the implementation of ray casting and the second one was the initialisation and rendering of multiple objects. I followed a tutorial to implement picking which could give the first object that was hit by the ray. But I could only draw a small number of voxels by drawing them explicitly. The next step was to generate multiple objects. In DirectX is called instancing. Following the tutorials from Frank Luna[43], rastertek[60] and Braynzarsoft [52], I failed to implement their techniques thus I abandoned the idea of instancing as I was discovering and reading about the concept of chunks. In the following product backlog, the second technique - generating an array of voxels by dragging the mouse - was abandoned but the generation of voxels with a mouse click was kept for further investigation.

The fourth iteration (see iteration 4, appendix C) involved the definition and optimisation of chunks [53]. All the objectives were met as an array of voxels could be rendered and performance tests were conducted to test the size the maximum size of a single chunk that the system can handle. This proved that optimisations had to be made on several aspects of rendering chunks as setting the size of a chunk to 10x10x10 = 1000 individual voxels or 12,000 triangles being rendered each frame required 2.2 Gigabytes of memory (see iteration 4, appendix B). This then led to a partial solution which decreased the number of triangles by 50%. Further future optimisations may include frustum culling, triangle merging and instancing.

The artefact does not meet the requirements of the specification as only the first objective of terrain rendering was partially met. With what I have learnt since the beginning of the project, I could have started development much earlier so that I would have allocated more time to set up the framework and try different techniques such as octaves. Optimisations would still have to be made to decrease bottlenecks and achieve better performance.

### Future works

* Future work
* The goal of each thesis is to help someone understand the research you have done on a topic and take that a step further

# References

[1] Tech radar. 2019. From ray tracing to AI: best gaming technology advances in the last decade. [ONLINE] Available at: <https://www.techradar.com/uk/news/from-ray-tracing-to-ai-best-gaming-technology-advances-in-the-last-decade>. [Accessed 5 May 2020].

[2] GDC. (2017). Believable Humans (In Video Game Animation). [Online Video]. 27 May 2017. Available from: <https://www.youtube.com/watch?v=Aj-zNjff7wY> [Accessed: 5 May 2020].

[3] Giantbomb. 2008. Destructible Environment. [ONLINE] Available at: <https://www.giantbomb.com/destructible-environment/3015-243/>. [Accessed 5 May 2020].

[4] Arcade museum. 1995. Asteroids. [ONLINE] Available at: <https://www.arcade-museum.com/game_detail.php?game_id=6939>. [Accessed 5 May 2020].

[5] Dobransky, M., 2017. *Efficient simulation of environment destruction in games*. 6. Prague: Charles University.

[6] dsvolition. 2001. Red faction. [ONLINE] Available at: <https://www.dsvolition.com/games/red-faction/>. [Accessed 5 May 2020].

[7]EA. 2008. FROSTBITE. [ONLINE] Available at: <https://www.ea.com/frostbite>. [Accessed 5 May 2020].

[8] Wikipedia. 2007. AnvilNext. [ONLINE] Available at: <https://en.wikipedia.org/wiki/AnvilNext>. [Accessed 5 May 2020].

[9] GamerNode. 2009. History 101: Destructible Environments in Videogames. [ONLINE] Available at: <http://www.gamernode.com/history-101-destructible-environments-in-videogames/>. [Accessed 5 May 2020].

[10] logicalincrements. 2020. Graphics Card Comparison Chart. [ONLINE] Available at: <https://www.logicalincrements.com/articles/graphicscardcomparison>. [Accessed 5 May 2020].

[11] Charvat, J., 2003. *Project Management methodologies: Selecting, Implementing, and Supporting Methodologies and Processes for Projects.* 1st ed. New Jersey: John Wiley & Sons. [https://books.google.co.uk/books?hl=en&lr=&id=Pd5AnH3ZA1AC&oi=fnd&pg=PR9&dq=project+planning+methodologies&ots=Q8DTHnChOH&sig=lQsukIRDSEzwA\_OwuSDm60hs9a0#v=onepage&q&f=false](https://books.google.co.uk/books?hl=en&lr=&id=Pd5AnH3ZA1AC&oi=fnd&pg=PR9&dq=project+planning+methodologies&ots=Q8DTHnChOH&sig=lQsukIRDSEzwA_OwuSDm60hs9a0" \l "v=onepage&q&f=false)

[12] LEWIS, J., 2011. *Project Planning, Scheduling & control*. 5th ed. Asheville, North Carolina: McGraw-Hill.

[13] projectmanager. 2015. The Ultimate Guide to… Project Planning. [ONLINE] Available at: <https://www.projectmanager.com/project-planning>. [Accessed 5 May 2020].

[14] zenkit. 2018. 7 Popular Project Management Methodologies And What They’re Best Suited For. [ONLINE] Available at: <https://zenkit.com/en/blog/7-popular-project-management-methodologies-and-what-theyre-best-suited-for/>. [Accessed 5 May 2020].

[15] content.intland. 2015. Why is Scrum so Popular? Why is Scrum So Successful? [ONLINE] Available at: <https://zenkit.com/en/blog/7-popular-project-management-methodologies-and-what-theyre-best-suited-for/>. [Accessed 5 May 2020].

[16] scrumguides. 2017. The Scrum Guide™. [ONLINE] Available at: <https://www.scrumguides.org/scrum-guide.html>. [Accessed 5 May 2020]. [17] Scrum of one, <https://www.raywenderlich.com/585-scrum-of-one-how-to-bring-scrum-into-your-one-person-operation>

[18] wikipedia. 1991. Scorched Earth (video game). [ONLINE] Available at: <https://en.wikipedia.org/wiki/Scorched_Earth_(video_game)>. [Accessed 5 May 2020].

[19] wikipedia. 1995. Worms (1995 video game). [ONLINE] Available at: <https://en.wikipedia.org/wiki/Worms_(1995_video_game)>. [Accessed 5 May 2020].

[20] Steam. 2015. Broforce. [ONLINE] Available at: <https://store.steampowered.com/app/274190/Broforce/>. [Accessed 5 May 2020].

[21] Steam. 2019. Noita. [ONLINE] Available at: <https://store.steampowered.com/app/881100/Noita/>. [Accessed 5 May 2020].

[15] ActiveCollab. 2017. Waterfall Project Management Methodology. [ONLINE] Available at: <https://activecollab.com/blog/project-management/waterfall-project-management-methodology>. [Accessed 5 May 2020].

[22] Gamasutra. 2011. Destruction. [ONLINE] Available at: <https://www.gamasutra.com/view/news/126940/Opinion_Destruction.php>. [Accessed 5 May 2020].

[23] Coumans, E., *Overview of destruction and dynamics methods,* presented at SIGGRAPGH 2011, 2011

[24] Steam. 2012. Counter-Strike: Global Offensive. [ONLINE] Available at: <https://store.steampowered.com/app/730/CounterStrike_Global_Offensive/>. [Accessed 5 May 2020].

[25] Fortune, S. A sweepline algorithm for Voronoi diagrams. *Algorithmica* **2,**153 (1987).

[26] Wikipedia. Voronoi Diagrams. [ONLINE] Available at: <https://en.wikipedia.org/wiki/Voronoi_diagram>. [Accessed 5 May 2020].

[27] H. Ledoux, "Computing the 3D Voronoi Diagram Robustly: An Easy Explanation," *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, Glamorgan, 2007, pp. 117-129.  
doi: 10.1109/ISVD.2007.10  
keywords: {computational geometry;3D Voronoi diagram;3D geometric computing;Delaunay tetrahedralization;Robustness;Algorithm design and analysis;Programming profession;Data mining;Geographic Information Systems;Distributed computing;Computational geometry;Sorting;Application software;Data structures},  
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4276112&isnumber=4276090>

[28] Maur, P., 2002. *Delaunay Triangulation in 3D*. Doctoral. Pilsen, Czech Republic: University of West Bohemia.

[29] wias-berlin. 2007. TetGen Features. [ONLINE] Available at: <https://wias-berlin.de/software/tetgen/features.html>. [Accessed 5 May 2020].

[30] Wikipedia. Fracture. [ONLINE] Available at: <https://wias-berlin.de/software/tetgen/features.html>. [Accessed 5 May 2020].

[31] Eric G. Parker and James F. O’Brien. Real-Time Deformation and Fracture in a Game environment. In Proceedings of the ACM SIGGRAPH/Euro graphics Symposium on Computer Animation 2009. Pp. 165-175.

[32] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins and Greg Turk. A finite element method for animating large viscoelastic flow. ACM Trans. Graph, 2007.

[33] James F. O'Brien and Jessica K. Hodgins. "Graphical Modeling and Animation of Brittle Fracture". In Proceedings of ACM SIGGRAPH 1999, pages 137–146. ACM Press/Addison-Wesley Publishing Co., August 1999.

[34] Crytek. CRYENGINE. [ONLINE] Available at: <https://www.crytek.com/cryengine>. [Accessed 5 May 2020].

[35] Noita steam page <https://store.steampowered.com/app/881100/Noita/> [21]

[36] Wilder, Michael W., "An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain" (2015).Honors Research Projects. 217.

[37] Lorensen, W. and Cline H., 1987. *Marching cubes: A high resolution 3D surface construction algorithm*. Corporate Reseach and Development. New York: General Electric Company.

[38] cs.carleton. 2020. An Implementation of the Marching Cubes Algorithm. [ONLINE] Available at: [http://www.cs.carleton.edu/cs\_comps/0405/shape/marching\_cubes.html#1](http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html" \l "1). [Accessed 5 May 2020].

[39] Matthew Fisher. graphics.stanford. 2014. Marching Cubes. [ONLINE] Available at: <https://graphics.stanford.edu/~mdfisher/MarchingCubes.html>. [Accessed 5 May 2020].

[40] Figure 1.8, PCGamer, 2013, <https://www.pcgamer.com/everquest-next-landmark-announced/> and GamerSpot, <https://www.gamespot.com/forums/system-wars-314159282/everquest-next-revealed-game-changer-29431034/> [Accessed 5 May 2020].

[41] Figure 1.8 Holz D, Beer T and Kuhlen T. Soil Deformation Models for Real-Time Simulation: A Hybrid Approach. In Workshop in Virtual Reality Interactions and Physical Simulation. The Euro graphics Association, 2009.

[42] Figure 1.7, 3d-map-generator. 2018. FROM HEIGHTMAP TO 3D TERRAIN IN 3 EASY STEPS. [ONLINE] Available at: <https://www.3d-map-generator.com/3d-map-generator-terrain-how-it-works/>. [Accessed 5 May 2020].

[43] Luna, F., 2010. *Introduction to 3D Game Programming with DirectX11*. 1st ed. Dulles, Virginia: David Pallai., p598-627,2010.

[44] Medium. 2016. Pixels and voxels, the long answer. [ONLINE] Available at: <https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190>. [Accessed 5 May 2020].

[45] Github. 2005. ImGui. [ONLINE] Available at: <https://github.com/ocornut/imgui>. [Accessed 5 May 2020].

[46] Wikipedia. Landmark (video game). [ONLINE] Available at: <https://en.wikipedia.org/wiki/Landmark_(video_game)>. [Accessed 5 May 2020].

[47] Minecraft. 2011. Minecraft. [ONLINE] Available at: <https://www.minecraft.net/en-us/>. [Accessed 5 May 2020].

[48] agilebusiness. 2011. Chapter 10: MoSCoW Prioritisation. [ONLINE] Available at: <https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation>. [Accessed 5 May 2020].

[49] visual-paradigm. 2011. What is Class Diagram?. [ONLINE] Available at: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>. [Accessed 5 May 2020].

[50] Test plan, “IEEE Standard for Software and System Test Documentation, in IEEE Std 829-2008, vol., no., pp.1-150, 18 July 2008

[51] Michael J. DickHeiser. C++ for Game Programmers, Second edition, Charles River Media, Boston, Massachussets, ISBN: 1-58450-452-8, 2007.

[52] Braynzar Soft. 2015. Picking. [ONLINE] Available at: https://www.braynzarsoft.net/viewtutorial/q16390-24-picking. [Accessed 14 April 2020].

[53] Let's Make a Voxel Engine. <https://sites.google.com/site/letsmakeavoxelengine/home/chunks> [Accessed 21 April 2020]

[54] Minecraft.2015. <https://www.minecraft.net/en-us/> [Accessed 07 May 2020]

[55] Wikipedia. 2015. Constructive solid geometry. [ONLINE] Available at: <https://en.wikipedia.org/wiki/Constructive_solid_geometry>. [Accessed 9 May 2020].

[56] docs.microsoft. 2019. Common Language Runtime (CLR) overview. [ONLINE] Available at: <https://docs.microsoft.com/en-us/dotnet/standard/clr>. [Accessed 10 May 2020].

[57] docs.oracle. 1993. Java Virtual Machine Technology. [ONLINE] Available at: <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/index.html>. [Accessed 10 May 2020].

[58] ecma-international. 2012. Standard ECMA-335 Common Language Infrastructure (CLI). [ONLINE] Available at: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>. [Accessed 10 May 2020].

[59] Atomontage. 2015. atomontage. [ONLINE] Available at: <https://www.atomontage.com/>. [Accessed 29 April 2020].

[60] Rastertek. 2016. Instancing. [ONLINE] Available at: <http://www.rastertek.com/dx11tut37.html>. [Accessed 12 May 2020].