APRIL 24, 2023

FINAL REPORT

TERRAFORGE: 3D TERRAIN BUILDER

WILLIAM KEARNEY-MITCHELL
2019310

KEYWORDS: DIGITAL TERRAIN, DIGITAL ART PROGRAM, 3D GRAPHICS, WORLDBUILDING, TECHNICAL ART, OPTIMISATION, SHADERS, GAME ENGINE, UNITY

Table of Contents

Abstract

People who create fictional worlds for mediums such as books, films, and video games lack a suitable program they can use to carve out the topology of their worlds in 3D. Such a program must be easy to use for artists and non-artists alike, as well as giving suitable tools and visuals to create a compelling and immersive 3D environment. TerraForge has been developed to fill this gap. Despite delays in development, the prototype shows great promise if developed further. The finished product should serve as an all-in-one tool that allows the easy creation and sharing of highly detailed islands and continents to assist in the worldbuilding process.

Background

"Worldbuilding is the process of constructing a world, originally an imaginary one, sometimes associated with a fictional universe."[1]

Worldbuilding is an incredibly widespread hobby that has been enjoyed for as long as humans have had imaginations. One need not look further than the explosive success of the *Star Wars* universe, or the world of *Lord of the Rings* to see its wide appeal. In the modern day, fictional worlds are created for mediums such as film, video games, books, or simply for their own sake.

Worldbuilding is a particularly popular hobby among players of Tabletop Roleplaying Games (TTRPGs). This medium allows the created worlds to be explored fully by players and serve as an opportunity for the designer to share their creation with others in a uniquely interactive space. This level of interaction obviously necessitates an incredibly fleshed out world, complete with information on individual cities, natural features, ruling factions and more.

Motivation

The time and care taken to sculpt these fictional locations is immense, and many digital tools exist to aid in the process. Huge binders have been replaced by online databases that keep track of each aspect of the world, and how they all interconnect. The process of designing the layout of the world has changed too, and while many people still prefer to ink out each island and continent, there now exist programs that attempt to facilitate this.

However, such software is well known by worldbuilders to be difficult to use, with a steep learning curve and a lack of features. During an interview with this project's client, he stated: *"I've found myself less inclined to make maps for my projects because current map-making software is unintuitive at times and it takes a long time to make something that I am happy with"*. Alongside other issues, one main feature absent from many popular map design tools is the ability to sculpt a landscape in 3D. Much of the detail present in a world can be lost when presented as a flat map, and there is a lot to be gained by allowing creators to carefully tune the hills, valleys, and lakes of their environments.

These issues are what TerraForge aims to tackle. The program is designed to be an easy-to-use, 3D landscape designing tool, with a focus on high-detail terrain sculpting. Users will be able to render country-scale locations with realistic textures and detailed geometry, as well as make use of a host of 3D assets such as trees, buildings, and roads to make unique and immersive maps.

Case Studies

To better understand the similar tools already available, a number of case-studies were done. Time was spent learning and analysing each one to determine how it works, what it does well, and what its flaws are. Some programs were map-makers similar to this project, others were programs that employed terrain manipulation techniques relevant to the early stages of development. Researching these existing tools helped to gather ideas for features to implement in this project, as well as identify weaknesses in the current ecosystem that would allow TerraForge to stand out by addressing. This process also helped to understand existing standards and common practice, so as to ensure any user migrating over to TerraForge from another tool would find many of the features familiar, and thus lessen the learning curve significantly.

In total 3 in-depth case-studies were carried out:

Roll20

Roll20 is an online tabletop roleplaying website that allows users to virtually share a ‘table’ with figurines, dice, and, of course, a map. As such, Roll20 features an in-depth 2D map creating tool to design dungeons, towns, and other places where players may need a physical representation of the space they’re in. As a case-study, this helped to get an idea of what things to avoid when designing the project. Creating a complex environment in Roll20 is difficult, and this is mostly due to the low number of free assets, and the difficulty in resizing and transforming them. The creative process feels rigid and the grid structure (while necessary for the type of games played) was restrictive. This case-study directly contributed to the decision to prioritise ease of use over complexity in TerraForge, as well as ensure all tools available to the user were intuitive, fluid, and felt good to use.

Planet Zoo

Planet Zoo was chosen for its in-depth terrain manipulation tools, and its incredibly polished feel. While being a game primarily about designing and managing a zoo, the game allows the user to distort and sculpt the plot of ground they are given using a suite of different tools. This case-study was used mostly as a reference for many of the more technical aspect of the project, from terraforming tools to UI. Having an industry standard example, similar to many aspects of TerraForge, allowed design decisions to be made that were already similar to what many users were used to. For example, by analysing the camera movement controls of Planet Zoo, the camera controls in TerraForge had a smoothly functioning example that they could emulate.

Inkarnate

Inkarnate is an online 2D map creation tool, specifically for worldbuilding and setting design. It boasts a wide array of tools to create impressive looking maps, such as: different layers for water and land, painting on land in different brush shapes, a stamp tool for placing multiple icons, and a wide range of textures and assets to choose from. While much of this program must be paid for to be used, the free version is a useful example of the status quo for world-building tools already available. This case-study informed the general direction the project needed to go in order to be similar enough to other tools out there so as to cater to people's needs, but also unique enough to offer something different to the programs already out there.

Client

Initially, this project was going to be informed by a large survey of many people who design and create worlds as a hobby. Unfortunately, due to ethical issues and time constraints, this became unrealistic. Instead, the project was overseen by one primary client, who served as the first port of call for user-acceptance testing and quality control. The client in question is a well-established worldbuilder, with experience creating for written work and video games[2].

To gather initial requirements and gain an insight into the specific issues surrounding map-creation, an interview was held with the client. While this does not have the same large-scale involvement that a survey would, perhaps resulting in a narrower view of the subject matter and consequently a less applicable program, having just one client allows for a much more fluid and rapid development process. The close proximity of the client also allows for a much more agile approach to development, as user testing can be done easily after each feature is implemented. This increased communication and capability for product evolution offsets the disadvantages of a downsized requirements gathering process, as without constant involvement from a client, many of these requirements would have run the risk of being miscommunicated or misunderstood regardless.

Initial Design

Following the gathering of requirements and using the case-studies as examples to iterate and improve on, a rough design was created. This included a paper mock-up of the UI, as well as diagrams illustrating how the most important features should function (figure 1).

The draft features designs for the toolbox, cursor, and options menu, as well as demonstrating how the UI interacts with the terrain. For example, it was decided early on that the cursor should be displayed flush to the terrain, conforming to its curves, to reduce any uncertainty of what area the user is selecting. Examples of the terraforming tools, object placement, and path creation were also included, each detailing which inputs were required from the user to achieve the desired effect.

Not only was this useful from a design standpoint, but also allowed the client to assess the draft before requirements were set out, further reducing the risk of miscommunication. Confident that the client and developer had the same vision, this then allowed a more concrete plan to be written up.

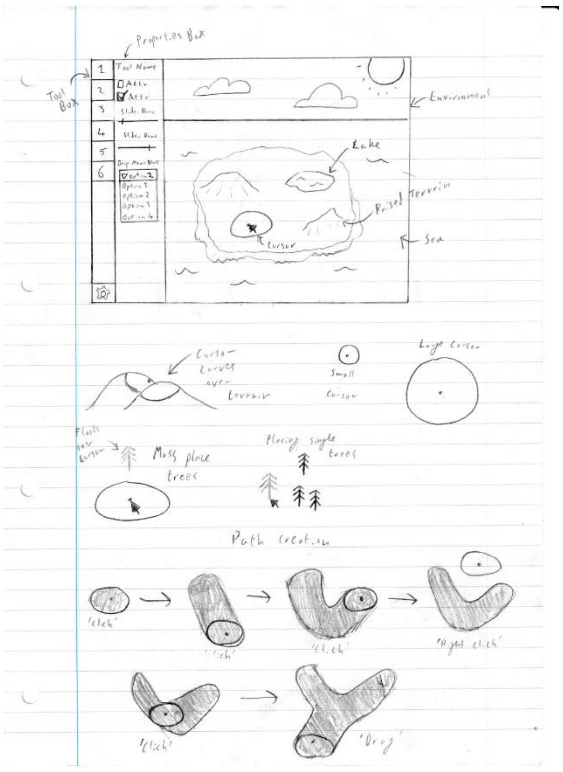

Fig 1. Design Draft

Development Methodology

Given that the project was developed by a very small team, it was well suited to a more agile methodology. This takes full advantage of the flexibility afforded by the ease of communication both within the development team and between the team and the client to provide rapid prototypes that can be assessed and improved with user-acceptance testing. This mirrors the approach that the Rapid Application Development (RAD) methodology takes[3], however in this project, it is combined with a decidedly lean approach. Lean project management focusses on reducing "waste"[4] and in the case of this project, waste comes in the form of unnecessary extra work. This extra work can come in the form of half-developed features that don't make it into the final product, or rewriting old code when a new feature must be implemented, both of which take up valuable development time and interrupt an otherwise streamlined development process.

To implement this Lean principle, two techniques were employed. Primarily, development closely followed the Minimum Viable Product (MVP) structure. In practice, this meant that each prototype of TerraForge after the first stage of development represented a fully usable program with no features partially implemented. Each new prototype then built upon the previous one, implementing one or two new features that were fully polished and ready to use. The advantage to this approach meant that delays in development could be very easily handled by cutting features from the final product with little impact to the rest of the software. Additionally, this facilitated a "pull" approach to development, in line with another principle of Lean project management, where a new feature would only be started when the previous one was fully completed, lessening the workload on the one-person team.

Another waste-reducing tactic employed was the focus on forward-planning. Part of developing each feature fully included "futureproofing" the code so that minimal rewrites were needed to implement later planned features on top of it. While this would lengthen initial development time when implementing the first requirements, it would be offset by the ease in which the codebase could be built upon in the future. However, this is a high-risk strategy as the assumption that the later features would make it into the final product is not a guaranteed one. This means that it was quite possible that a large amount of work would be done to plan for non-existent features, unnecessarily lengthening development time and creating waste as opposed to reducing it. This risk was evaluated and considered to be worth taking however, as the consequences of having to work on a poorly planned codebase would be severe if the project reached that point in development, which it was planned to.

Development Plan

Development will be done using the Unity game engine. This is done to mitigate the overhead of setting up a 3D rendering pipeline and take advantage of Unity's available out-of-the-box tools. Importantly, this will allow more time to be spent on developing the features requested by the client, and less time on unnecessarily redesigning tools that are readily available.

To facilitate an MVP approach, requirements were split into 4 phases, with each phase being dependant on the full completion of the last, but the requirements within each phase being independent. This allowed a lot of flexibility to decide which feature to develop next, while also providing the structure necessary to plan for the future.

These requirements were set out in a Design Document and are as follows, and the dependencies can be seen in Figure 2. While care was taken to set out success criteria for each to provide concrete tests to determine when a feature was fully implemented, they could be further improved by using SMART (Specific, Measurable, Achievable, Relevant, Time-bound) objectives to give extra structure to assist in development and further facilitate good client-team communication.

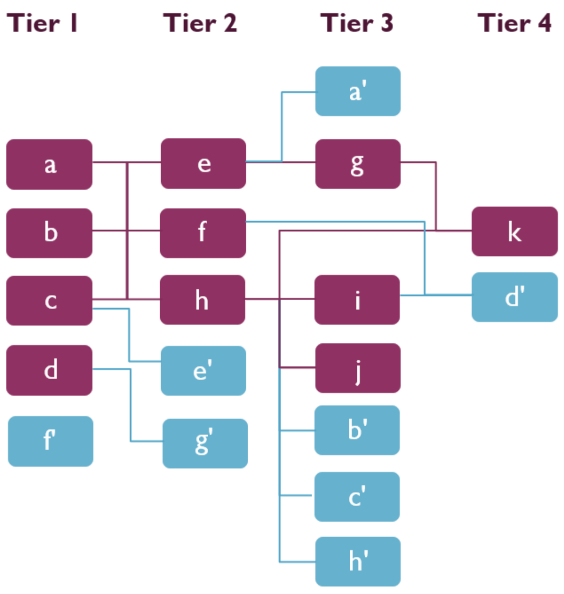

Fig 2. Requirement Dependence Chart (extensions highlighted in blue)

The structure of the requirements also provides natural points to test the software at. Testing against success criteria was done after the implementation of each feature, alongside more rigorous user acceptance testing at the end of each phase. This allowed for the client to make suggestions for changes once a phase was complete, but before the next one began, and it became "locked in". This ensured that the software stayed within the client's expectations. One possible improvement could have been the use of user acceptance tests after each requirement, instead of after each phase. This would have taken full advantage of the on-site client for continuous feedback but, on the other hand, could have perhaps resulted in an increase in scope-creep.

Phase 1

Requirement A: Brush Implementation

- Cursor follows mouse and sticks to surface of terrain, following variations in height
- Cursor grows/shrinks when the user uses the scroll wheel in conjunction with the ctrl button, to a maximum and minimum size
- Cursor shows some indication when the user clicks/holds the left click
- Cursor isn't displayed when interacting with menus

Requirement B: Toolbox

- Toolbox is displayed in the top left corner of the screen, and maintains its position in different aspect ratios

- Different icons can be displayed in each section of the toolbox
- Icon changes to indicate when hovered over or selected
- When a tool is selected, allow the possibility of a propert9ies box extending from the right of the toolbox
- Allow checkboxes, sliders, drop-down menus, and information to be displayed in the properties box
- Display a floating information 'tooltip' when hovering over a tool in the toolbox

Requirement C: Blank world

- Create a height-map backed mesh that starts with a circular area of positive values in the centre
- Create a translucent mesh overlaying the environment at a constant height to represent the ocean
- Allow the user to pan around the environment using the middle mouse button
- Allow the user to scroll in and out, to a minimum and maximum zoom level
- Allow the user to rotate around the point in the centre of the view using the right mouse button

Requirement D: Skybox and Textures

- Obtain textures for the different terrain types that tile well and can be applied to the land-mesh
- Texture the ocean with scrolling waves using normal maps
- Texture the ocean in such a way that it appears darker the further away the sea floor is
- Obtain a skybox containing a blue sky and sun and apply it to the environment

Phase 2

Requirement E: Basic terraforming

- Implement 2 tools in the toolbox, one called raise terrain, and one called lower terrain, with appropriate tooltips
- Add properties tabs to each of these tools with sliders for intensity and cursor size
- Increase the values in the heightmap within the cursor when the user left clicks with the raise tool, with the intensity decreasing with distance from the centre of the circle
- Decrease the values in the heightmap within the cursor when the user left clicks with the lower tool, with the intensity decreasing with distance from the centre of the circle
- Implement a minimum and maximum height that can be obtained using these tools

Requirement F: Terrain painting

- Implement a terrain painting tool in the toolbox, with a tooltip, sliders for cursor size & intensity, and a drop-down menu for the different terrain types (grass, dirt, stone, sand, snow, swamp)
- Apply the appropriate texture to the land mesh within the cursor when the user left clicks with this tool, blending into the existing texture around the edges of the cursor

Requirement H: Single object placement

- Implement a single object placement tool in the toolbox, with a tooltip, size slider, and dropdown menu including some prototype objects (simple house, tree, bush, cactus)
- Display a ‘ghost’ version of this object on the ground where the user’s cursor is pointed when this tool is selected, conforming to the terrain height
- Place the object in the environment permanently in the exact location the ghost was located when the user left clicks
- Create an option in the properties tab that toggles whether the object is positioned at the angle of the terrain, or pointing straight upward regardless
- Move the position of any relevant object when terraforming around them

Phase 3

Requirement G: Water placement

- Prevent ocean appearing when lowing inland terrain
- Add ripples to the surface of the ocean
- Implement a water placement tool in the toolbox, with a tooltip and a dropdown menu for rough/smooth water
- With this tool selected, allow users to select the wall of a dip in terrain, and if there are no objects in the way, highlight the perimeter of the dip that water would fill
- Prevent the user placing water if the area is too large
- If the area is valid, allow the user to place a body of water with left click
- The water should have ripples if ‘rough’ was selected, and be calm otherwise
- Create another tool to facilitate the removal of placed water
- Prevent any changes to terrain around or inside the body of water

Requirement I: Object distribution

- Create a new tool with the same properties as the single object placement tool, but with additional sliders for area size and density
- When the user hovers over the environment with this tool selected, show the cursor with a hovering ghost of the object selected above it
- When the user presses or holds left click, place several objects within the cursor so as to fill the space without clipping objects into each other
- The objects should have random positions and rotations
- The objects should not clip into already present objects
- The objects should follow all the rules applying to single object placement

Requirement J: Object removal

- Create a new tool with a tooltip and checkbox that toggles mass object removal or individual object removal
- With individual object removal the user should be able to hover over an existing object and it have a tint to indicate it is selected
- With mass object removal selected, the cursor should be visible with a size slider in the properties box, and all objects selected by it should be tinted
- When the user left clicks, any object selected should be removed from the environment

Phase 4

Requirement K: Road & river placement

- Create 2 new tools with tooltips and dropdown menus for different types of road/river as well as a size slider
- When the user hovers over the environment with this tool, display the road/river texture inside the cursor
- If the user left clicks in valid area (no objects, no water for roads) create the start of the road/river and allow the user to extend from it by dragging the mouse
- Allow the user to left click to set new points to the road/river, each new segment extending from the last
- Allow the user to start a new road/river by right-clicking
- If a new road/river is started on top of an existing one, make a branch
- If a river starts or ends at the ocean or a lake, connect it to the body of water
- The road/river should follow the height and angle of the terrain
- If a road intersects a river, place a bridge in the style of the road

Extensions

Requirement A': Advanced terraforming

- Create new terraforming tools with tooltips and the same properties as raise/lower
- The flatten tool raises and lowers terrain to meet the point originally selected
- The mountain tool raises terrain in a sharp prism shape, the orientation of which follows the cursor
- The plateau tool sharply raises terrain to meet the point originally selected, leaving higher terrain alone
- The carve tool is the inverse of the plateau tool
- The roughen tool will randomly raise and lower terrain in its area

Requirement B': Animated features

- Apply animations to some of the placed objects such as trees and bushes

Requirement C': Complex animated fauna

- Add fauna patches to the list of placeable objects and give them complex animations
- Such fauna could include shoals of fish, sheep, vultures, birds, polar bears etc.

Requirement D': Automatic terrain décor

- Modify the terrain painter so that it removes terrain specific objects when it is laid down, and randomly adds more where it is painted

Requirement E': Starting environments

- Give the user the option of starting with a completely ocean map, the default circular island, or a random starting configuration

Requirement F': Clouds

- Add translucent random clouds that float across the environment
- Allow the user to change the density of the clouds by using the world-settings tool

Requirement G’: Time of day

- Allow the user to change the time of say using a special world-settings tool
- Have the ‘sun’ move across the sky depending on the time of day, and the light level reflect that change
- Have some objects such as buildings display lights when it is dark
- Simulate a sunset when the time is at dawn or dusk

Requirement H’: Increased asset pool

- Increase the number of objects available to the user
- Increase the number of terrain types available to the user

Limitations

While TerraForge aims to solve many problems that contemporary map-making software has, it cannot facilitate the huge scope the hobby has to offer. One such limitation is one of scale, TerraForge is designed to work best at an island scale. This means that the terrain will look the best when users are sculpting features such as mountains and rivers. While the program is usable at other resolutions, it lacks the necessary tools for extreme detail on the scale of individual towns or meadows, or at incredibly large scales such as entire planets. This is not beyond the future scope of the project however, as with an increased asset pool and more terraforming tools, this would be possible. More notably, this software is not designed for city building, as the tools provided are more suited to natural terrain. Given the way the program is written, it would be very difficult, if not impossible to implement this kind of feature and would better be suited to its own project in and of itself.

Additionally, while there is no limit to the imagination of humans, these limits do apply to TerraForge. At the moment there is no support for more extreme landscape features (rivers of lava, radioactive pools, obsidian towers, ect.), however, TerraForge is the kind of software that would greatly benefit from user-created features. It is quite possible, in the future, to allow users to upload their own models and textures for their unique worlds and implement these exact landmarks. Allowing this kind of collaboration from users is also a fantastic way to offload a lot of work that would be unrealistic for one developer to achieve, while also building a strong supporting community for the program.

Development Timeline

Initially, the first tier of requirements took longer than expected to implement. This was mostly due to the decision to pay very close attention to their implementation, making sure any further features would be easy to implement on top of them. This was key to the flow of the project, since mistakes and sloppy coding early on would cost a lot of time further down the line, since every future requirement depended on and implemented the first tier.

Camera Controls

For example, a long time was spent redesigning and fine tuning the camera control script. This was the module that allowed the user to pan, rotate and zoom the camera around the environment and the main issue was deciding how the camera would interact with the terrain itself when colliding. Many different approaches were taken, tackling issues such as whether the user should be able to move the camera out of bounds, where the camera would end up when colliding with or moving past terrain, and how the camera dealt with extreme zoom and awkward angles.

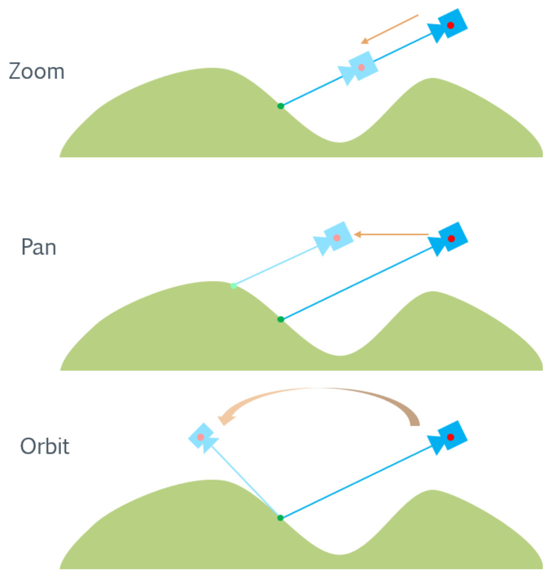

Fig 3. Zooming, panning, and orbiting are done with the scroll wheel, right mouse button, and middle mouse button

The final design is too complex to cover in its entirety, but involves the camera being paired with a point on the ground it's facing towards. The pair would then interact differently depending on what transformation was taking place (for example rotation fixes the point and orbits the camera around it, while panning moves the camera on the x and z axes while the point follows). Collision with the ground was handled by moving the camera up in the y-axis until it was no longer inside the geometry, and zooming was clamped between a maximum and minimum value. An example of this can be seen in Figure 3.

The reason so much care was taken with this design, is that as development progressed, the geometry of the environment would only get more complex and taxing for a badly designed camera control system. Perfecting the controls early on ensured that it could handle even the most complex terrain, and extra time would not need to be spent further down the line tweaking this module constantly to keep up.

Mesh Generation

Another early aspect of development that was key to get right was the mesh generation algorithm for the terrain. A badly designed script would cause performance issues, strange bugs, and headaches that would only worsen with the introduction of more complex features later on. Clearly a fast, intuitive solution needed to be found before implementing anything else on top of it.

This algorithm went through many prototypes and consisted of two main parts: land mesh generation, and sea mesh generation. Firstly, the land mesh needed to be generated according to a number of parameters such as: size, resolution, and an internal height map. Before arriving at the final solution, different approaches were implemented and tested to determine their speed and efficacy. The first, naïve approach generated a new point and triangle matrix each time the height map was updated, with one vertex for each entry in the height map. This was slow and unnecessary, and a simple improvement was to only update the vertex array since the triangles would always stay the same.

Another approach attempted to simplify the mesh to allow for a higher resolution while staying under Unity's vertex limit[5]. This involved identifying regions in the height map with the same elevation and covering them all with a large quad instead of many smaller ones. This approach simplified the mesh considerably, however the extra processing power required to make this simplification caused it to be slower than other approaches. This was tested with and without vertex duplication, but even without having to worry about duplicating vertices, it was still significantly slower.

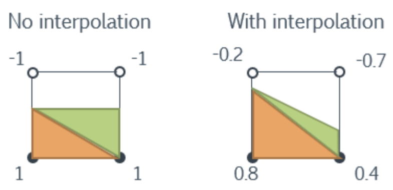

Fig 4. Triangles drawn in a cell using the marching squares algorithm

The sea mesh generation algorithm was much more complex than the land mesh script, since it involved the use of the Marching Squares algorithm to contour the water's edge around the terrain. The Marching Squares algorithm is a very common technique used when generating a mesh based on a grid of values, where each value is classed as either inside or outside the mesh. The mesh is split into squares, with 4 data points on each corner. These data points determine what triangles are drawn within this square to make up that portion of the mesh. By allowing each datapoint to take on a float value based on how far inside/outside the mesh it is, the positions of the vertices can be interpolated between them, resulting in a smoother mesh[6]. For our case, these values are the height of the terrain above or below sea level. See figures 4 & 5 for more details.

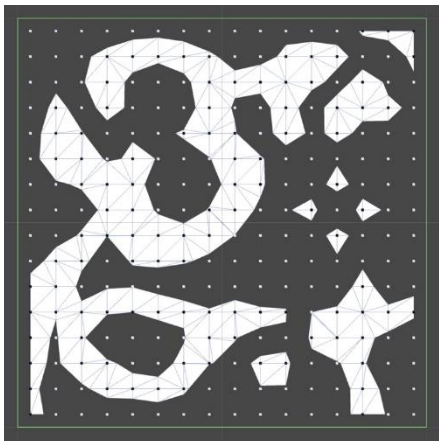

Fig 5. A large mesh drawn using multiple cells

The mesh simplification algorithm was tested on the water mesh and provided a significant speedup against the constant resolution algorithm. However, both scripts still proved to be slow and limited in terms of the mesh resolution they could provide.

The final solution consisted of a chunk-based parallelised approach using compute-shaders. By splitting up both the land and sea mesh into 'chunks' of fixed size, the vertex limit was circumvented. This allowed for the sea mesh to be computed using a less vertex-efficient "constant-resolution algorithm with vertex duplication" that could be computed in parallel, allowing for a significant speedup. This approach had the added advantage of allowing only the necessary parts of the terrain to be updated when the user interacted with it using a tool. By drawing a bounding box around the cursor and only updating the chunks that intersected with it, terraforming was sped up considerably as less updates were being made per frame.

Unfortunately, the introduction of chunks led to further problems. Notably the calculated vertex normals, used to calculate shading, for each chunk did not consider the neighbouring vertices that were part of a different chunk. This led to noticeable seams at the boundary between each chunk, where the shading was not smoothly interpolated. By creating a custom shader that displayed the normal vector of each pixel in the mesh, this problem was identified and solved by manually "sewing up" each seam, providing the vertices with the necessary information about their neighbouring chunks to calculate the correct set of normals. This effect can be seen in figure 6.

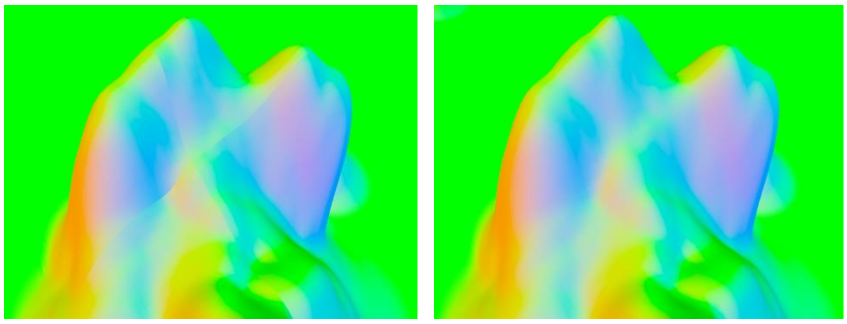
Fig 6. Seams between chunks (left) have been smoothed out (right) and visualised using a custom shader

In hindsight, further improvements could have been made to the mesh algorithms. One major bottleneck was the need to generate a new physics collider for the land mesh every time it was updated, this was used to stop the camera accidentally intersecting it but was computationally much more expensive than simply generating a visual mesh. Some way to avoid having to create a collider for the ground mesh, or generating a simplified collider would have gone a long way in terms of optimisation. Additionally, there was no consideration when it came to implementing LOD (Levels of Detail) in the terrain. If this project were to be developed further, and more decoration and textures added, LOD would have to be implemented to reduce the quality of assets further away from the viewer, in order to preserve graphics processing speed[7].

Realistic Water

In order to achieve the realistic looking water required, some technical work was needed. Firstly, the water plane was given a translucent blue material, with the exact hue being tied to the terrain height map. Using this, the colour was interpolated between a dark blue and a lighter, more transparent blue depending on the depth of the water. The client examined many different combinations of colours and advised the team on the one to use for the final product, opting for a less vibrant but more realistic looking colour.

To achieve the effects of waves, a common technique was utilised. Instead of displacing any vertices in the sea mesh, two normal maps were scrolled across the surface in opposing directions. This was done using a shader, so the angle and speed of each normal map could be modified. Since normal maps control how light reflects off an object, scrolling these across the surface of the water gave the illusion of waves, causing the light to bounce and reflect in different directions over time. The end result was a convincing approximation of calm waves while keeping the toll on performance to a minimum, as underneath the effects the water mesh remained a flat plane.

Given more work and a more experienced team, it would be possible to research more advanced methods of simulated waves. A very simple improvement could use stochastic texture sampling to eliminate the "tiled" look of the water when viewed from a distance[8]. More realistic results can be obtained by perturbing the vertices of the sea mesh using sinusoids, and other advanced techniques beyond the scope of this paper can be used to further improve the look and behaviour of the waves[9]. These techniques can be seen in games such as *Hydrophobia*[10] and *Sea of Thieves*[11], which simulate extremely realistic water in real time for the purposes of player-immersion.

User Interface

Due to Unity's inbuilt tools, it was incredibly easy to construct a simple UI toolbox inspired by the case-studies and connect it to the functionality of the terraforming tools. Creating a cursor that conformed to the terrain was slightly harder, however. The first solution was to create a flat canvas that a cursor sprite could be placed in front of, with the canvas textured with a ground texture. Using an orthographic camera pointing at the canvas, a texture could be created from what it could see. This texture could finally then be applied to the ground mesh to give the illusion of the cursor wrapping around the curves of the terrain.

Moving the cursor to the location of the user's mouse involved casting a ray from the main camera to the ground mesh, then converting those coordinates to coordinates on the canvas, and transforming the cursor sprite there, which would in turn appear to move across the ground mesh (see figure 7 for more details). This process was obviously incredibly involved, but most importantly, it meant that the resolution of the ground texture and the cursor were constrained by the resolution of the camera that was viewing the canvas. This led to pixilated looking terrain and a low-resolution cursor, so instead another solution was implemented.

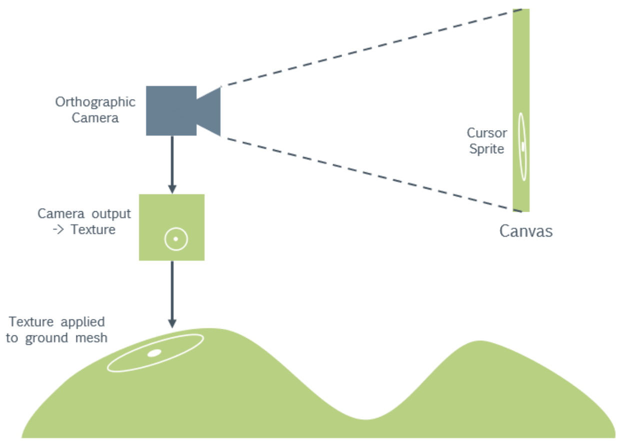

Fig 7. Initial sprite drawing technique

The second solution involved creating a custom shader that would draw the terrain texture directly onto the terrain mesh, and then additively blend the cursor sprite on top based on coordinates passed into the shader. This approach meant that more data had to be send to the GPU, a slow process, but allowed for a much higher resolution texture for the ground, as well as making it trivial to modify to accommodate the chunking approach that was later used.

Terraforming Tools

To allow the user to manipulate the shape of terrain to their will, a number of intuitive terraforming tools had to be created. It was important to not create so many that the user would be overwhelmed with choice, but also have enough distinct tools to fulfil each use case so the user was always able to achieve the desired effect. In the end, 7 tools were settled on:

- Raise/lower terrain: Smoothly raises or lowers terrain in a circle around the cursor for as long as the user holds down the action button
- Flatten terrain: Stores the height of the terrain when the user clicks the action button, and will then lower all terrain the user moves over gradually to that level
- Plateau terrain: Stores the height of the terrain when the user clicks the action button, and will then raise all terrain the user moves over gradually to that level
- Equalise terrain: A combination of flatten and plateau tools that brings all terrain the user selects to the same height
- Roughen terrain: Creates random bumps in the terrain around the cursor
- Smooth terrain: Reduces sharp edges and bumps in the terrain to create smooth areas

The first 5 tools are 3D extrapolations of linear functions, using the distance from the centre of the cursor as the input. The raise and lower tools are therefore based on cosine waves, whereas the next 3 have more complex functions (see figures 8 & 9 for details). Each tool has a size and intensity slider that changes the radius of the cursor and the strength the tool in question will use. Cursor size can also be changed using the mouse wheel and the control button as a shortcut.

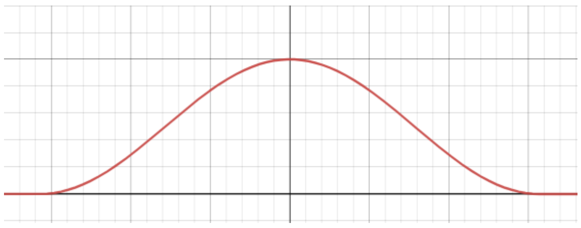
Fig 8. Raise terrain function, x-axis represents distance from cursor

The roughen tool uses 2D Perlin noise to raise and lower sections of the terrain within the cursor's area. Perlin noise is a type of multi-dimensional random noise characterised by a lack of sharp gradients, allowing it to be used to generate organic looking effects[12]. The same section of noise is sampled for as long as the user is holding down the action button, but by releasing and pressing it again, a new seed is chosen. The smooth tool uses a simple unweighted box-blur algorithm to average the heights of all neighbours of a point and interpolate between a point's original height and the new average height while the user holds down the action button. In retrospect, more time could have been devoted to researching the most suitable blurring algorithm to use for this, as well as taking more care with the other tools to write more intuitive algorithms.

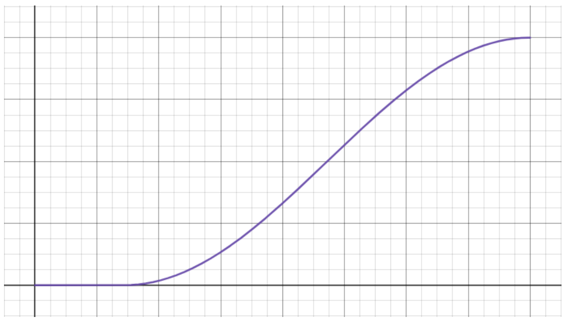
Fig 9. Flatten function, x-axis represents distance from cursor

Terrain Painting

After the implementation of terraforming tools, the next feature to add was the ability to paint different types of terrain onto the ground. The user would be able to choose which areas of the map were snowy, rocky, sandy, etcetera. The terrain would be placed down at the cursor location, becoming lower in opacity as it reached the edges, smoothly blending in with the surrounding terrain. Usefully the size and intensity attributes could also be applied to this tool, with a low intensity allowing for a certain terrain type to be blended very lightly with another. This blending of terrains provided many more options for the user than just the ones listed, as sandy rocks could be created by lightly brushing the rock texture onto sand and so on.

This was achieved by creating a texture array object. Each texture in the array served as a mask for each terrain type, with the pixel values representing the opacity of each terrain type when drawn onto the ground mesh. This method allowed for a much smaller delivery of data to the GPU, which would then calculate the colours of each pixel in the resultant ground texture by reading each mask and sampling from each different terrain texture type. This approach also allowed for the mixing of terrain types using alpha blending. For example, consider a pixel with mask values of (1, 0, 0) meaning it is 100% grass, 0% sand, and 0% rock. If the user were to use the terrain painting tool lightly on the area using a sand brush, the resulting mask values may become (0.6, 0.4, 0), meaning 60% grass, 40% sand, and 0% rock. Note that the resultant mask values always add to an opacity of 1, with any value added to one mask detracting from the values of the others.

Had the project received more development time, the appearance of the different terrain types could have been improved significantly. For example, realistic grass could be added to the grass terrain type using techniques such as GPU instancing with flat sprites, and normal maps could be implemented to provide additional detail to the otherwise flat surfaces.

This feature firmly proves the usefulness of shaders and GPU processing when designing projects such as this. Not only can it improve the performance of graphics related features, such as the combination of different terrain textures pixel-by-pixel, that would be incredibly slow to perform on the CPU, but other parallelisable tasks can also be offloaded onto it. This was used throughout the project to speed up vertex-by-vertex calculations when applying the Marching Squares algorithm and could have easily been extended to deal with calculating the effects of terraforming tools in parallel.

Automatic Terrain Painting

A feature specifically requested by the client during development was for a system that would automatically change the type of terrain based on the height of the ground. This was intended to be useful when initially designing the topology of the terrain, and then turned off so the user could paint their own textures later on. This feature was accepted into the development cycle as it provided a lot of value to the project, at little time cost and was in an acceptable margin in terms of scope creep.

While easy to implement and not otherwise noteworthy, the addition of the feature speaks to the ease of communication between the client and developer working to produce a higher quality product that would have otherwise been achieved if the client were not involved. And while this does increase the risk of scope creep, open discussion between both teams allows for compromises to be made and end up increasing the client's trust and patience.

Obstacles and Complications

Unfortunately, despite the considerable care taken to forecast development times and plan a schedule around that, many setbacks were encountered during the development process. One assumption that contributed to this was the planned 16-hour work week that the forecasting was based on. While this was possible in the early stages of development, as different projects took time away from developing TerraForge, and team moral lowered, it became unreasonable to maintain such a time commitment to a single project. This obviously had an impact on the pace in which new requirements were completed, and even though many were finished in less time than originally forecast thanks to the principle of forward planning, development took a considerable hit.

Thanks to the strict adherence to the MVP model, any delays in development were easily mitigated by reducing the scope of the project. As each requirement consisted of its own independent package, the least important ones could be cut from the final product without affecting the rest of feature implementation. Unfortunately, while this was initially acceptable, more and more features ended up being cut from the final product, from extensions to tier 4 & 3 requirements. Consequently, the final product contained less and less of what the client initially requested. Even though a complete final product could still be delivered by the deadline, it was unfortunately very lightweight and lacked the depth promised.

This reinforces the idea of the *iron triangle*, a concept which states that improving one of the following: scope, time, or cost, will always come at the expense of the other two[13]. In this example, by constraining time and cost, it has come at the expense of the scope of the project. If the decision was made by the client to extend the deadline to allow more features to be implemented, this would then detract from the timeliness and cost. This decision is up to the client to make, and in this example, the time constraint was the top priority.

Other issues included slight underestimates when forecasting time taken to complete each requirement. This was simply due to the relative inexperience of the team and will improve with time. Similarly, a significant amount of time was taken to research techniques and methods to achieve effects. Had the team had the experience it has now, this time would have been greatly reduced and thus features developed more rapidly. While these were not risks that could have been eliminated, the planning stage could have included ways to mitigate them through more time allowance and deliberate time overestimates.

Appraisal

As stated, the use of MVP served to greatly reduce the severity of the multiple development setbacks. Had the project been developed using a standard waterfall technique, with little room for changes in scope, there would not have been a finished product to show the client by the end of the development timeline. It is for this reason that the agile/lean approach was well suited to this project, allowing an inexperienced team the flexibility it needed in order to work to the best of its ability, and still produce a quality product.

Conversely, the forward-planning approach worked slightly in opposition to this. While it is always beneficial to be wary of future work when developing a code-base, too much deliberation risks a significant slowdown, without the concrete promise of payoff down the line. This was the case with this project, as much of the work spent to optimise earlier features to further work went to waste, as the additional features could not be implemented into the final product. For example, while the water effect could have been achieved using a flat plane instead of a custom mesh, this approach would have not worked once the ability to add custom bodies of water was implemented. This is because the user would not have been able to create an in-land valley that dipped below water level, which was a concrete part of both the client and development team's vision. Additionally, implementing a marching squares algorithm early on would have allowed it to be reused later when the shapes of custom lakes were being calculated. The end result of this is that all of the work done to implement and optimise the marching squares algorithm was not strictly necessary for the final product. However, if the project were to be picked up again in the future and the later features added, this would prove invaluable.

This outcome highlights an important debate when it comes to developing software. Is it more efficient to take time to optimise and plan for the future while developing, or is it better to focus on the task at hand, completing it as much as is necessary and allowing the flow of development to continue? It could be argued that for a project such as this, who's approach was more agile and who's end result was more unclear, that spending so much time optimising and planning for future features handicapped the flow of development unnecessarily, and that that approach is more suited to static development methodologies such as waterfall. It's unclear what a better approach would have looked like, however more time should have been spent prior to development, devising a methodology that would better merge the principles of MVP and forward planning so that they worked with, rather than against each other. It is quite possible that both approaches are incompatible, and therefore a new methodology would have had to be adopted.

Reflection

Looking back on the decisions made when managing this project gives an insight into how it could have been possibly improved. As stated throughout the rest of this document, there were a multitude of individual development and management decisions that could have been altered to produce a better final product. Looking at the big picture, however, the management of the project, as well as the premise and approach, were largely satisfactory. Other than the obvious problem of time management and forecasting, the process of working on the project, from conception and planning to development and testing, was largely painless and free flowing. The work breakdown structure (WBS) created to keep track of which requirements should be handled in what order, as well as the list of success criteria made it easy to focus on the task at hand, as well as knowing what the next steps were after each package was signed off. This approach would have scaled well to a larger team too, as each developer would be able to work independently of the others until the time came to sign off the entire tier.

The project remains to be one of the only examples of a 3D map creator for use by worldbuilders, and still very much has the capacity to fill that niche. TerraForge's tools and visuals are unique in the industry and will only improve given more time.

Further Work

As stated before, the state of the code-base makes further work on TerraForge very easy. Thanks to the principle of forward-planning, the project may be picked up even years down the line and finished to a satisfactory degree. Additionally, this particular style of software means it is ripe for the inclusion of user-created content. If TerraForge were to ever be released, it would be beneficial to include a feature that allowed users to upload their own models and textures so they could tailor the program to be optimal for the exact world they are designing.

The inclusion of this feature would also help to establish a strong community around the software, where users can discuss and share their creations, expanding and deepening TerraForge far beyond the ability of any one individual. This type of community collaboration has occurred countless times in the past with different software and video games. For example, art software such as *Procreate* allows users to create and share their own digital brushes[14]. This means that when artists are creating a specific effect, such as fur or wood, they have access to custom made brushes made by the community for that specific purpose.

If developed even further, features not included in the original design could be implemented. Such features could include different export options, such as 3D printable models. Users would then be able to physically show off and use the maps they design if they had the right equipment to print them. One significant upgrade that could be considered would be the move from a 2D height map to a fully 3D volume map. This was not considered viable in the original specification and therefore would require an entire rewrite to implement, but it would allow previously impossible features like overhangs and archways to be sculpted using a marching cubes algorithm. This algorithm functions very similarly to the marching squares algorithm discussed earlier, but involves constructing a 3D mesh out of cells, each of which take on a predetermined configuration based on the "insidedness" of the 8 points on the cell's corners[15].

The amount of polish a project like this can receive is almost unlimited. So, there will always be room to improve the textures, add extra shaders, and improve the realism of the scene. Such additions could take the form of realistic looking clouds, atmospheric refraction alongside sunrises and sunsets, heat distortion in the desert and snow particles in the tundra. Care must be taken to ensure these aesthetic additions do not become too expensive to render in real time, since the focus must always be on the usability and versatility of the tools, rather than hyper-realistic but slow visuals.

Conclusion

TerraForge was a project with high aspirations and an interesting premise that unfortunately fell short of its promised scope due to a variety of factors. Nevertheless, it remains to be a highly optimised and well-functioning final product with enough features to fulfil its original intended purpose. Thanks to the design philosophy of MVP and forward-planning that was carried out through the entire development process, setbacks were handled elegantly and schedules could be reworked on the fly.

The state of the final product leaves a lot of room for further work, made easy by the careful consideration taken in the initial stages of development, and it is quite possible that TerraForge could create a strong user community given the time needed to flesh out the remaining features. It is hoped that, in the future, TerraForge is allowed to reach its full potential as an intuitive 3D map design tool for the worldbuilding community.

Citations

[1]. Hamilton, *You Write It: Science Fiction*. ABDO, 2009.
[2]. "Roxirinart" Twitter. https://twitter.com/Roxirinart (accessed Apr. 24, 2023).
[3]. S. Mcconnell, *Rapid development : taming wild software schedules*. Redmond (Washington): Microsoft Press, 2014.
[4] E. Ries, "The Lean Startup | Methodology," Theleanstartup.com, 2019. http://theleanstartup.com/principles (accessed Nov. 22, 2022).
[5] U. Technologies, "Unity - Scripting API: Mesh.indexFormat," docs.unity3d.com. https://docs.unity3d.com/ScriptReference/Mesh-indexFormat.html (accessed Nov. 22, 2022).
[6] D. Ramalho, "Marching Squares," David's Raging Nexus, Dec. 18, 2022. https://ragingnexus.com/creative-code-lab/experiments/algorithms-marching-squares/ (accessed Apr. 27, 2023).
[7] D. P. Luebke, *Level of detail for 3D graphics*. Amsterdam Morgan Kaufmann [20]05.
[8] "Procedural Stochastic Texturing in Unity," *Unity Blog*. https://blog.unity.com/technology/procedural-stochastic-texturing-in-unity (accessed Apr. 28, 2023).
[9] E. Darles, B. Crespin, D. Ghazanfarpour, and J. C. Gonzato, "A Survey of Ocean Simulation and Rendering Techniques in Computer Graphics," *Computer Graphics Forum*, vol. 30, no. 1, pp. 43–60, Oct. 2010, doi: https://doi.org/10.1111/j.1467-8659.2010.01828.x.
[10] "Hydrophobia Water Physics," *www.youtube.com*. https://www.youtube.com/watch?v=UQ0HDe2-R4Q (accessed Apr. 27, 2023).
[11] "Sea of Thieves - Water Physics," *www.youtube.com*.
[12] A. K. Ginting, K. Sari, C. Fadhilah, R. N. Yusra, D. Hartama, and M. Zarlis, "Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game," *Journal of Physics: Conference Series*, vol. 1255, p. 012064, Aug. 2019, doi: https://doi.org/10.1088/1742-6596/1255/1/012064.
https://www.youtube.com/watch?v=9UAn3bDDx_U (accessed Apr. 27, 2023).
[13] Prince2, "Project Management Triangle: A Triple Constraints Overview | USA," *www.prince2.com*, 2019. https://www.prince2.com/uk/blog/project-triangle-constraints
[14] "Import and Share - Procreate® Handbook," *Procreate*. https://procreate.com/handbook/procreate/brushes/brushes-share/ (accessed Apr. 28, 2023).
[15] "Polygonising a scalar field (Marching Cubes)," *paulbourke.net*. http://paulbourke.net/geometry/polygonise/

Appendix

Software Demo Screenshots