DIGWORLD

[aka. the game formerly known as “runworld” and/or “treasure hunt”]

Liam Esparraguera, Luke Shannon

ABSTRACT

DIGWORLD is a first-person, 3D platformer video game with a focus on engaging player movement in a dynamic and modifiable environment. Core features of the game include procedural generation, real-time environment geometry manipulation, and momentum-based movement and character control. The objective of the game is to collect as many randomly-spawning ‘points’ as possible while traversing the game’s 3D environment, achieved by means of using the ‘terraforming’ controls to modify the environment – adding and deleting terrain at will – as one is navigating it. In this regard, a core motivation for our development of the game was to create a unique system of player-to-game-world interactions that are inherently fun to engage with and evoke a sense of novelty. As a result, DIGWORLD features several mechanics of technical interest, notably: procedural generation of a 3D environment using fractal noise / fractal brownian motion, use of the Marching Cubes algorithm to generate mesh geometry from an underlying 3D scalar field, and real-time modification of environmental mesh geometry by means of chunking. The player input of DIGWORLD adheres to a traditional standard for traditional keyboard-based movement controls: the WASD keys control player movement direction with respect to the current first-person view direction, which is controlled by the mouse. In addition, the ‘Q’ key is used to bring up a birds-eye minimap view, which helps the player locate themselves in relation to ‘points’ around the map, and the left ‘Shift’ key triggers a grappling hook, which allows the player to propel themselves about the map using the environment geometry.

GAME CONTROLS

W A S D - Move	**Q - Map**	**Left Shift - Grapple Hook**
Mouse - View	**Left Click - Build**	**Right Click - Destroy**

*NB - Due to issues with mouse locking on Safari, we strongly recommend that you access and play our game demo on Google Chrome.

INTRODUCTION

GOALS

Our core goal for this project was to create a game with mechanics that both personally inspired us and made use of some of the techniques and systems we found most interesting over the course of COS 426. For this reason, we found ourselves interested in procedural generation at the very start of our brainstorming project; Luke had prior experience with procedural generation algorithms through his generative art practice, and Liam had engaged with procedural environments in forays into the development of small video games in the "rogue-like" genre. Nevertheless, neither of us had created a system for generating 3D environments before, and so we found this to be an alluring challenge for ourselves, especially to do so using the noise-based generation techniques introduced in the *Book of Shaders* readings for Assignment 3 (Gonzalez Vivo & Lowe). Inspired by the work of Sebastian Lague (more on this in the next section), our group settled on implementing the Marching Cubes algorithm as a means by which to generate the geometry for our terrain from an underlying noise-derived scalar field, and, upon considering the importance that the environment was to play in our development focus, decided that the best means to match this focus would be to design gameplay mechanics that centered on interaction with the player's immediate environment. In our own personal experience with video games and interactive media, we both find ourselves most inspired by games with unique mechanics that, through the novelty and polish of their interactions, can create fun player engagement even beyond the scope of challenge-reward dynamics; on this subject, a text on video game mechanic design, *Game Feel* by Steve Swink, was particularly influential and inspired the grapple-hook movement mechanic featured in our game. We hope that our game demo can, with its terraforming, dynamic environments, and movement mechanics, accomplish both this goal of facilitating player-driven, sandbox gameplay and stand as a technical achievement within the expectations of this course.

PREVIOUS WORK

Since our primary initial motivation in this project was to explore the procedural generation of 3D game environments, we looked to existing video games with procedurally-generated worlds in order to see the possibilities afforded by existing techniques. In this regard, some pieces of related work that we found especially successful and influential were:

- Minecraft, for the sophistication of its gradient noise-derived procedural terrain.

source: History of world generation – Minecraft Wiki

- Terraria, for the manner in which movement mechanics, especially its grappling hooks, interact with the procedural environment and open up new possibilities for dynamic player movement.

source: How to Make a Grappling Hook in Terraria | DiamondLobby

- No Man's Sky, for its ray-based mining into a mesh environment

source: https://nomanssky-archive.fandom.com/wiki/Mining

Synthesizing our favorite components of these three games – the gradient-noise derived environments of Minecraft, the terrain-based traversal of Terraria, and the smooth, 3D mesh terraforming of No Man's Sky – would be our ultimate goal for this project. In researching the feasibility of this goal, our group found the work of Sebastian Lague, a prominent computer graphics / game development Youtube creator, particularly inspiring as, in a pair of videos entitled *Terraforming* and *Marching Cubes* (see citations), Lague creates a basic terraforming video game out of these three components on a scale that we felt would be achievable, if ambitious, for this COS 426 project. However, Lague did have access to the provided tools and systems of the Unity game engine, and so our approach to implementation would prove quite important to our project...

APPROACH

Inspired by the high-level structure of Sebastian Lague's *Terraforming* project, we constructed our base approach to this project by organizing our work into two distinct focuses, which by extension became the major technical components of our game.

1. Terrain Generation
    a. Scalar Field Generation
    b. Marching Cubes Implementation
    c. Mesh Coloring / Shading
    d. Chunking

    e. Terraforming
2. Player Control
    a. Input Listening
    b. Camera Control
    c. Collision Detection
    d. Movement + Acceleration
    e. Grappling Hook Controls

With this framework in mind, we could separate the major components of our game into discrete sections (the sub-lists above) of reasonable size/complexity that could be implemented fairly separately and, when assembled together, create complex behaviors. We hoped that this approach would allow our work to be highly parallelizable, which was important since both of us would need to operate on quite different and busy schedules throughout the project-development period. Furthermore, we anticipated that this fairly-modular approach would lend itself nicely to more rapid debugging, which would be necessary for our two-person group to make efficient progress. Regarding the scope of our game, we were cognizant of the fact that, especially as a two-person group, there would be a necessary tradeoff between the scale and complexity of our project; with regards these limitations, we decided to prioritize the demonstration of interesting and creative computer graphics and game development techniques, and hope that this approach would yield a successful final product.

TECHNICAL METHODOLOGY

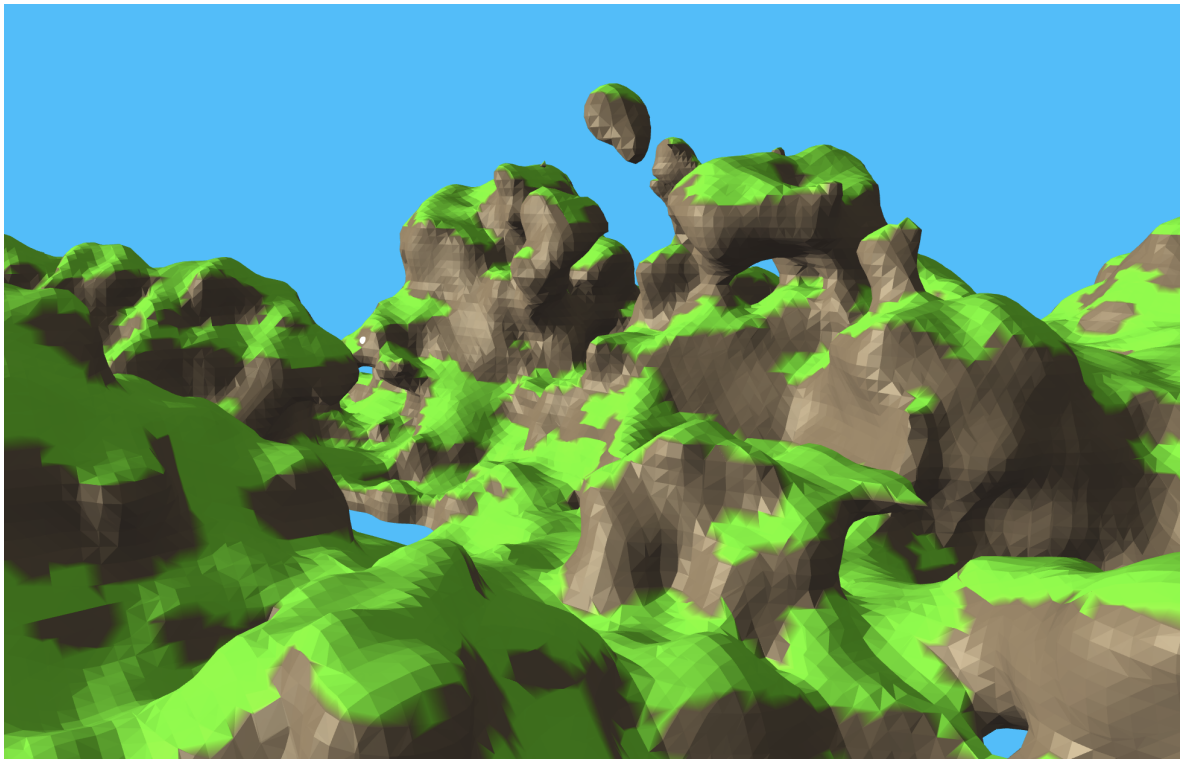
TERRAIN GENERATION

I. PROCEDURAL GENERATION

Our approach to procedural generation, as informed by the techniques of Minecraft, Terraria, and other popular video games, was centered around the use of pseudo-random noise generation which, when integrated with other numerical offsets, could yield a 3D scalar field of significant interest. This scalar field, when coupled with a threshold value against which values at points in space can be compared, could then be used to distinguish between which points in space are “land”, meaning within solid terrain, and “air”, meaning outside of terrain. With this ability, the Marching Cubes algorithm (next section) can be used to generate 3D mesh geometry.

In technical terms, our procedural generation is performed by generating a 3D scalar field using a combination of fractal noise (fractal brownian motion) and a linear

offset based on the point coordinates. We generate our fractal noise using 4 octaves of simplex noise, and find that, with a gain factor inversely proportional to the frequency, this creates terrain that is reasonably smooth and at scale, but also contains a visually-engaging amount of fine details. Our 'linear offset' is an extra term that is linearly proportional to the point's y-coordinate so as to weight the generated team towards lower y-values, thus limiting the number of floating islands and creating solid ground on which the player can stand. This is the fundamental model for our scalar field generation; it is quite simple but, with some fine tuning for the parameter values, gives some lovely results.

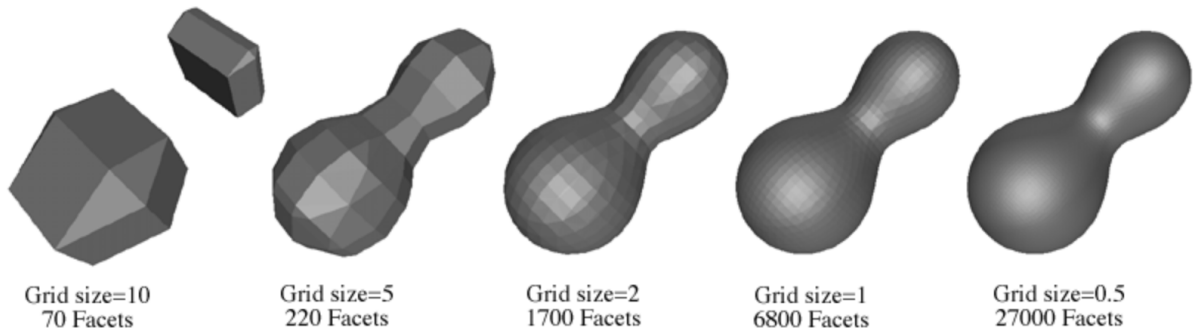


There are, of course, many systems and varied techniques for procedural generation including, but not limited to, Genetic Algorithms, Markov Chains, Cellular Automata, Voronoi Diagrams, rule-based procedural generation systems, and many more. When considering potential techniques for our world's procedural generation, we decided on our simplex noise-derived approach because it is best suited for mimicking natural landscapes, and is a very appropriate and efficient method for creating large scalar fields to use with the Marching Cubes algorithm to generate mesh geometry. Other systems may have been viable as well, and could serve for interesting future explorations of maps based on things other than the natural landscape.

II. MARCHING CUBES

Our terrain generation relies on the Marching Cubes algorithm to construct 3D mesh geometry from our noise-derived scalar field. Put simply, the Marching Cubes algorithm reconstructs a 3D surface from scalar data in a 3D space – commonly, to construct a surface from volumetric datasets or mathematical scalar field function – by sampling the volume on a regular 3D grid and determining patches of planar surface that best approximate the underlying isosurface, defined by the underlying scalar values and a threshold 'isolevel' (Bourke). In Marching Cubes, a cube is the underlying pattern on which the scalar values are sampled and, since a cube has 8 corners, this results in 2^8 = 256 potential combinations of which corners' sampled values surpass the isolevel. These cube geometry mapped to by these combinations are stored in a large lookup table, and a simple algorithm calculates the index of the correct cube as determined by the corner positions' values by using bitwise OR operations on powers of two corresponding to particular corners of the cube. Our implementation directly adapts this logic from the description of the Marching Cubes algorithm published by Paul Bourke (Bourke).

After the appropriate cube has been determined, the exact position values of vertices are determined by linearly interpolating along the cube's edges in proportion to the scalar field values at their adjacent vertices, and finally the vertices are converted to world coordinates with a simply scalar multiplication. This logic is all contained within the regenerateChunk() method of our Chunk class, and helper methods are define for vertex interpolation (interpolateVert()) and cube index calculation (getCubeIndex()). Compared to other methods of mesh generation, Marching Cubes is quite a fast algorithm, as it primarily uses direct arithmetic, bitwise OR operations, and static lookup tables to generate geometry. When implementing terrain manipulation ('terraforming'), we were, in all honesty, quite scared that our geometry generation would not be fast enough; however, we were pleasantly surprised to find that, with reasonable chunking, our Marching Cubes implementation could allow for real-time terrain regeneration with very little framerate slowdown in our local environment (eg. Liam and Luke's laptops).



source: http://paulbourke.net/geometry/polygonise/

III. MESH GENERATION + VERTEX COLORING

After vertex positions have been generated by the Marching Cubes algorithm, the next step is to construct a mesh which can be added to and rendered within a Three.js scene. To do this, we found it necessary to use Three.js's BufferGeometry, which is a "representation of mesh, line, or point geometry" that includes vertex positions, face indices, normals, colors, UVs, and more within a buffer (Three.js). This buffer storage reduces the cost of passing geometry data to the GPU, and thus offers a potential performance boost to rendering. In our research, this was the direct means available in Three.js of constructing a Geometry from an array of vertex positions, and so it was the option we went with by default. With some simple adaptations/optimizations – primarily, merging vertices that were incredibly close (eg. a floating-point calculation error) apart into a single vertex – we were able to have mesh generation that took only ~one second on page load to initialize the full map terrain.

In order to generate visual interest and replicate the appearance of hilly outdoor terrain a bit more, we used a property of Three.js's BufferGeometry to manually assign each vertex a color, such that section terrain that are not too 'steep', and so could pick up sunlight, appear grassy, and other, steeper sections are dark brown dirt. We implemented this by iterating over each vertex in the terrain's geometry and calculating a 'steepness' heuristic by taking the dot product between the vertex normal and the scene's 'up' vector (unit vector in positive y axis); if this dot product was above a certain threshold (0.6 in our final version), the vector would be assigned a green color, otherwise brown (Lague). This approach provided aesthetically pleasing, if simplistic, results; however, its true advantage/value is that, by assigning vertex colors based on geometry, the terrain mesh's material is able to respond to real-time updates in geometry as manipulated by the player.

IV. EDGE CONDITIONS

Our base implementation of scalar field generation, however, does not consider the edges of the map. Because fractal brownian motion is continuous, the map edge would have open mesh edges, as it assumes connectivity to its neighboring sections. We can fix this and close the mesh by decreasing the scalar field values at the edges within a fall off distance, thus ensuring that the field at these areas is beneath the threshold for solid terrain. Reducing the field linearly at the edges of the map to air means that the transition between air and land must occur before the edge of the map, and therefore be closed under marching cubes. This implementation patches up the 'open sides' of our map; previously, if the player were to out-of-bounds, they could see beneath the terrain and view the non-rendered back faces of our terrain mesh, but now the terrain is closed, and one cannot see the insides from any exterior view.

TERRAFORMING

I. CHUNKING

In order to permit engaging terrain manipulation by the user, terrain geometry must be able to be regenerated in real-time by the user, ideally on each frame on which a change to the terrain's underlying scalar field is made. Due to the scale of the game map, we found that regeneration of the entire map geometry was far too inefficient for any sort of real-time player control, and highly inefficient. If, as per our game design, the player is to be able to manipulate terrain only in one local area at a time – an appropriate analogy is the 'soft brush' from Assignment 0 – then geometry regeneration need not occur anywhere but in that local area. Thus, we found a solution to this issue in chunking, as suggested in Sebastian Lague's *Terraforming* video (Lague). Chunking is the process by which our game's terrain is generated in discrete sections that are then stitched together so as to seem a single whole. Our game implements chunking by integrating all of our geometry generation code into a Chunk class, which is then instantiated many times as children of our parent Terrain class; the Terrain class initializes a scalar field for the entire map, so as to ensure continuity between chunks (the edges of chunks can read into adjacent chunk's region of the scalar field), and then renders Chunks as child meshes in a grid, with X, Y, and Z offsets to their coordinate spaces. After implementing this approach, we found ourselves more confident that terraforming could be done efficiently.

II. RAYCASTING

For our terraforming mechanic, we wanted the player to be able to build at the point they were looking at, in a fashion similar to Minecraft and other first-person games. In order to do this, we needed to be able to find the intersection point of a ray that is cast from the center of the camera along the view direction and the landscape – we found that a well-suited technique for this is raycasting against the mesh, for which we could use Three.js's Raycaster class. In order to implement this raycasting, we intersect a ray from the center of the Player controller's camera (see player control section) into the terrain mesh itself, and read the position of the first intersection point found, as it is the closest to the player). Originally, we anticipated that this approach would be too slow, and so devised an alternative method of casting a ray and approximating its intersection by directly accessing the underlying scalar field of the map; however, this proved unnecessary, as our implementation using Raycaster() to cast the ray into mesh geometry proved fast enough in practice for a high framerate.

III. SCALAR FIELD MODIFICATION

In order to actually trigger regeneration of the terrain geometry around a point selected by the user, two things must occur: the scalar field itself must be modified, and the chunks needing update must be identified and have their geometry regenerated. The first issue was fairly direct: we simply access the scalar field at the nearest integer coordinates to the player-cast intersection point, and then iterate over a cubic volume around that point. In this cubic area, we decrease/increase the value of each value in our scalar field proportionally to the distance between that point and the center point to add/delete terrain, and this distance-falloff naturally results in a spherical shape for terrain manipulation. The second issue, of identifying which chunks correspond to the regions of the scalar field modified, is also fairly straightforward: we convert our intersection's world coordinates to 'chunk coordinates' by dividing by the global scale multiplier and taking the floor of the result and keep track of all chunks seen in a Set. Then, for each chunk in the set of affected chunks, we remove its mesh, call its regenerateGeometry() method, and add it back to the scene.

PLAYER CONTROL

I. INPUT LISTENING

Each movement command is initiated by a keystroke or mouse click, so by listening to these events directly through the document we can have a responsive app without significant background cost. We tie each user interaction to the player class in order to change boolean variables that queue different movements when we update the player. It is important to also bind these variable changes to the release of those keys, to turn off those interactions (for example, stop the player from walking forward).

II. CAMERA CONTROL

The camera control is mostly handled by the provided PointerLock class used for first person control in Three.js scenes. We lock the user's mouse on click such that they can use it to move the camera and navigate the world, and can exit to normal interaction by pressing the 'Escape' key. We used this Three.js game example for this camera movement scheme. The exception to this rule is if the player is looking at the minimap, at which point the camera moves to the top of the world to provide a birds-eye view of the map (see Other Gameplay Components).

III. COLLISION DETECTION

We implemented a custom collision detection ruleset and scheme using raycasting. Because our world uses our Marching Cubes implementation, the world is mostly continuous and consists of the mesh of each chunk, which can include smooth, unpredictable, and curving slopes. In order to detect if a player is on the ground, we raycast down from the player's position (where their head is) perform this procedure:

1. If there is an intersection within the length of the player's height, then they are on the ground.
2. If this distance-to-ground is less than the player's height, then we change their position to be their height above the point of intersection. Therefore, if for example a player builds terrain underneath their feet or walks on a slope, then their feet are slightly under the ground, and on the next frame they are moved upwards to place them the proper distance from the mesh.
3. If there is no nearby intersection, then the player is in the air and we continue to update their movement using their velocity and gravity. We also disable their ability to jump, as they are not on the ground.

Overall, this is a similar scheme to what we use when raycasting to find the point the player is looking at. We also consider the player's position as they approach the edge of the map. To prevent the player from falling off the map, we introduce world borders by clamping the player's position to within the edges of the map, within reason.

IV. MOVEMENT + ACCELERATION

It was important to us that our movement implementation work based on acceleration, so that a player could be rewarded with consistent/smooth movement by increased movement speed. We keep track of the player's position and velocity as persistent variables, and then add to the velocity when we update the player's WASD movement and gravity as acceleration values. To keep the player's movement consistent across different and variable frame rates and machine capabilities we scale the acceleration by time elapsed between frames instead of adding a constant amount each frame.

We also used our collision detection to change a player's movement. In this situation, the update order is important in order to maintain a consistent camera view. If collision detection happens before player movement, then the camera may shake as the player's position is alternately moved by forces like gravity and corrected by collisions.

V. GRAPPLING HOOK

The grappling hook is a simple feature, but one that we find to be very engaging thanks to the manner in which it functions in coordination with our terraforming, movement, and collision systems. Our implementation of grappling is simple - should the player initiate using the grapple hook (by pressing 'Left Shift'), we store the position of the current view intersection (should there exist one) and, on each frame, apply an acceleration in the direction of the vector from the current player position to the grapple point. In order to help prevent clipping into geometry, as well as grant the ability to 'swing' in addition to grapple in a direct line, we do not apply the grapple acceleration ot the player if they are within a certain distance (a GRAPPLE_LENGTH constant) to the grapple point. Surprisingly, this feature was implemented in a very short amount of time as a fun addition directly after we finished our final presentation of the project; we find that the fact that this momentum-based grappling hook system was able to be implemented with no changes to existing collision or movement code is testament to the robustness of our collision and control systems.

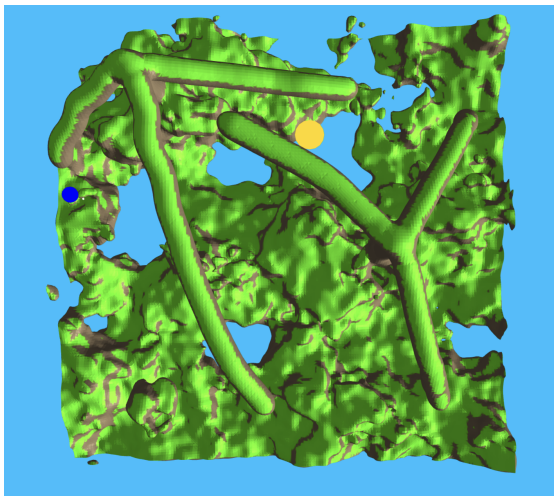
OTHER GAMEPLAY COMPONENTS

I. POINTS

The goal of the game is to find and collect points as quickly as possible. The points spawn randomly on the map as yellow spheres, and are collected by moving the player to within the radius of the sphere. When we update the player we check if the player is within range to collect the point, and if they are we update the score and generate a new point randomly on the map. By using the same sphere for the points we are able to keep object references to a minimum, even as the game time increases indefinitely.

II. MINIMAP

In order to locate points more quickly, we found it useful to use a second view from the top of the map. From this angle you can find the location of the point relative to your own position (the blue sphere). When the player presses q we move the camera to the top of the world, and when they release it we return it to the player's position. This has the added benefit of allowing us to view player creations on the map from an additional angle.



RESULTS + DISCUSSION

MEASURING SUCCESS

We measure success primarily by user engagement and how well the final project aligned with our initial goals. We were ready for terraforming to be a significant challenge, and therefore prepared ourselves for several tricky implementation challenges. By the playability of our final product, we consider ourselves successful in implementing our technical features to a degree stable enough to support gameplay. As a game itself, we find our project engaging and fun in a tactile sense for reason of its mechanics – we would often find ourselves playing the game for longer than intended when debugging – and, with respect to our motivations for the gameplay, we find that this is a significant success.

RECONSIDERATIONS + HINDSIGHT

It's hard to say what might have been a better path for this project. One thing that surprised us was how simple and fun it is to be able to manipulate terrain, and how much freedom it gives the player. Perhaps our game mechanic (collecting points) could have been expanded to work in more ways to be creative with this ability to terraform and create. For example, we think that this mechanic lends itself particularly well to an open-world game, but unfortunately with the time constraints we found it infeasible to try to make a persistent large scale world that would maintain user creations. One game model we have come up with, but a bit too late to be incorporated into the final project, is a race-against-time model: collecting a point resets a brief timer which constantly runs down and, if reaching 0.0, ends the game. In this game mode, the player would attempt to survive as long as possible, shaping the environment around them over time into more and more erratic forms.

FOLLOW-UP WORK

Mostly, our follow-up work would consist of expanding the project into new directions beyond what we attempted for this scope. That said, there is some tuning that could be done on collision detection–at high speeds or unusual terrain conditions, the player can occasionally clip into the surroundings. This could be addressed by checking the player's position against the scalar field and moving them upwards if the value indicates they are underground.

CONCLUSION

EVALUATION OF GOALS

Overall, we are very satisfied with this project. Because of the time constraints, our initial goal was to make an engaging and well-developed game around a single unusual mechanic. By this metric, DIGWORLD is a success. This mechanic, terraforming, also brought us to face a number of technical challenges, like procedural generation, collision detection, real-time modification of the environment, and chunking optimizations. This is a game that we actually enjoy playing, and while the scope is limited, it is not hard to imagine how this project might be expanded upon to become a fully-fledged sandbox or open-world game.

NEXT STEPS / FUTURE OF PROJECT

We would love to continue development on this project! We are both artistically-inclined individuals, and find a lot of inspiration in the creative results that can be achieved with the combination of computing and artistry. Learning Three.js, although a bit tricky, over the course of COS 426 has been a great experience, and we look forward to future projects we can build with this technology and all the other skills and techniques developed during this semester.

Future ideas we have brainstormed for this project include: the use of a custom fragmentation shader for pixel-by-pixel terrain coloring (independent of complexity of geometry), the creation of a water shader, continuous terrain generation (eg. world expansion), and additional movement mechanics, like gliding.

NOTED ISSUES + BUGS

- LOD Terraforming

  Unfortunately, the regeneration of map terrain at a lower level of detail does cause complications which often break our terraforming feature, as the overall map scale is reduced as an unintended result. Thankfully, we believe that we have found the culprit for this issue, which has to do with improper scaling between standardized and world coordinates in our scalar field generation; we have attempted to fix this in the short time before the assignment deadline but, since modifying the scale calculation significantly alters our procedural generation results (which we are

quite fond of in their current state), we thought it best to maintain our current terrain generation, and pursue this as a site for further development.

- Safari Mouse Locking
  Thanks to complications with mouse locking as controlled by Three.js's PointerLockControls on the Safari browser, our game is not easily playable in Safari; right-clicking to delete terrain interrupts the canvas with a menu prompt, and the control state is disturbed. We hope to look into browser behavior with mouse events in the future to fix this.
- Collisions with Rapid Movement
  Very rapid movement, as a result of the player moving faster into terrain than the ray cast can intercept it, has a tendency to allow the player to clip into the map geometry. This isn't a major issue with our current demo, but it is something we hope to address should we ever want faster in-air movement in the future.

CONTRIBUTION BREAKDOWN

NOTE

Both members of this project contributed significantly to all components of its design and development; there are few, if any, features that were developed in isolation by one person, and consistent communication and peer-coding were key to this project's efficiency. Nevertheless, certain areas of the code were subject to particular focus by one member or the other, and are presented below.

LUKE SHANNON - player control, movement, collision detection, points, terraforming, minimap

LIAM ESPARRAGUERA - terrain generation, marching cubes, chunking, grapple hook, terraforming

WORKS CITED

LAGUE, SEBASTIAN *Terraforming* , *Marching Cubes*

BOURKE, PAUL *Polygonizing a scalar field*

GONZALEZ VIVO, PATRICIO & LOWE, JEN *The Book of Shaders; 12-13*

MRDOOB *Three JS PointerLock Example*

COS 426 *Lecture Materials*