Robert Grier
6/7/2024

Demo Video:
https://youtu.be/L6XEXuVbLBU

Research Project: Terrain & World Builder

Introduction

A game engine can enable the design of a detailed game world by providing interactive tools for editing the layout of objects in the scene. The 3D terrain is the primary "object" in many outdoor scenes and requires specialized processing for its geometry and texturing. By allowing the terrain to be edited with a set of brushes, similar to brushes found in image editing tools, a game developer can carefully craft the environment as part of the game engine workflow. My project allows the terrain to be sculpted, painted, and decorated with foliage with responsive brushes. When coupled with the existing scene editing tools, a detailed game world can be iteratively developed.

Terrain Mesh

A terrain mesh can be constructed as a grid of vertices forming a plane. The vertices can then have various heights to break up the plane and form terrain features like mountains or valleys. This mesh can be constructed from a heightmap image where each pixel in the image dictates the height at that vertex.

Storing the terrain mesh as a traditional 3D mesh asset, where the vertex data is stored as a vertex buffer of 3D coordinates, is possible when the terrain does not change. However, this storage method limits the ability to make large scale changes to the terrain data at runtime, because the edited vertex buffer has to be reloaded onto the GPU. This is especially slow when the terrain is large and contains millions of vertices.

To enable real-time editing, the terrain vertex buffer only contains the 2D positions of the vertices, which are constants. The height at each vertex is stored as a texture called the "heightfield". In the vertex shader, the heightfield is sampled and the read value is used as the height value of the terrain. The advantage of this approach is that we can use shaders to edit this texture on the GPU, which is extremely fast and has immediate visual feedback for the small cost of sampling the heightfield texture in the vertex shader.

The game engine provides an interface for generating, saving, and loading terrain assets. An asset can be generated from a heightmap image or as a flat plane ready to be edited. When saved, all data needed to recreate the terrain is serialized into a Protbuf file. This binary format is useful over a standard image format because we can add game specific properties and metadata to the terrain asset. This single-file asset can then entirely recreate the terrain mesh, textures, and foliage. The asset is stored independently from the scene file. This allows the scene file to remain as pure text that merely references the binary terrain asset.

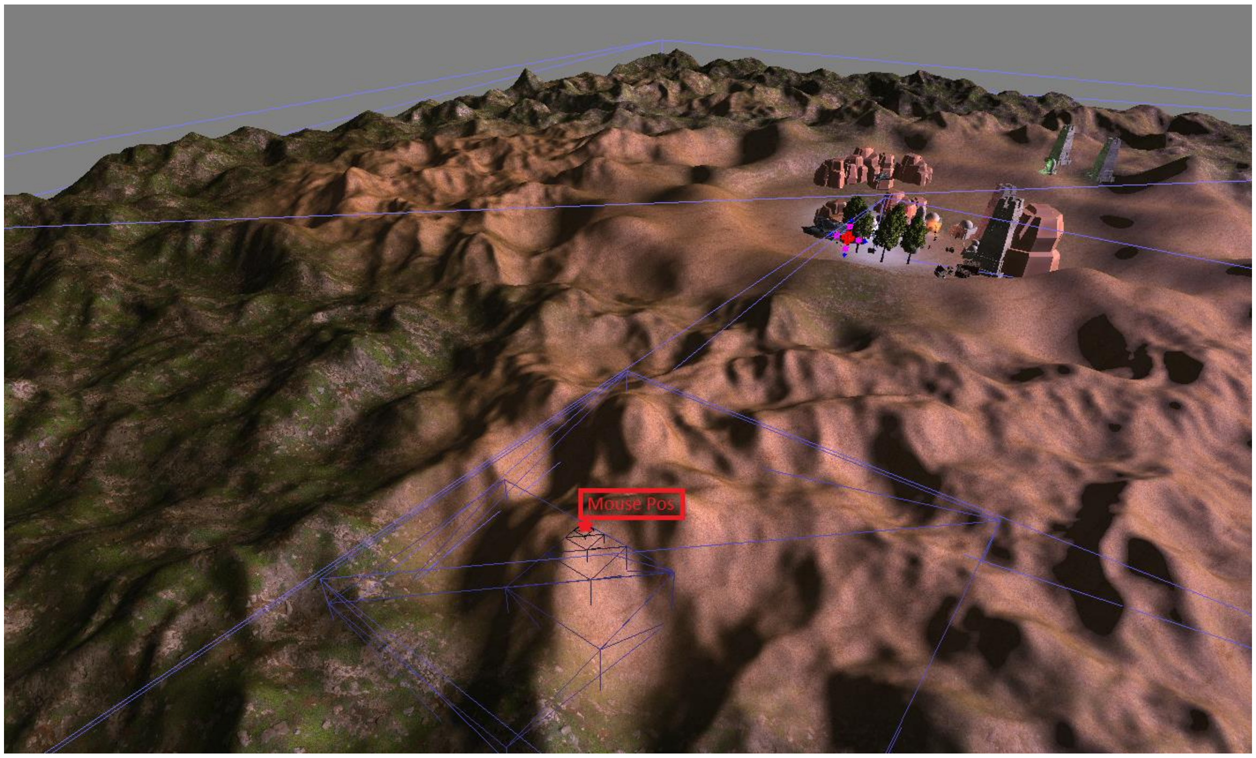
Quad Tree

A hidden but essential component of the terrain system is the quad tree. This data structure is a tree of nodes where each node covers a rectangular area of the terrain. Each composite node has four children that break up the area into quadrants. The tree has a maximum depth which is composed of leaf nodes. This depth is chosen so that the leaf nodes are reasonably small. All nodes have a constant 2D rectangular area, but they also store the minimum and maximum height of all terrain vertices that fall within their area. This allows an axis-aligned bounding-box to be constructed around all parts of the terrain contained within a node.

The primary purpose of the quad tree is to enable efficient intersection testing with the terrain using logarithmic, early-out algorithms. The most useful test in the engine context is a raycast, such as a raycast from the mouse position into 3D space. The raycast algorithm performs a ray vs AABB test starting with the root node, and continuing with the children for any test that returns true. Tests that do not need to be performed are skipped, and all leaf nodes that the ray intersects with are returned in a list. Then, for each leaf hit, the ray is tested against the terrain triangles within that leaf. This is still fast when the leaves are small and contain few triangles. The ray vs triangle test is optimized to return the distance from the ray origin to the intersection point as an additional output. This allows the final part of the algorithm to simply return the closest intersection point.

The quad tree is interesting because it requires a tradeoff between memory usage and speed in a way that is not merely hypothetical. Adjusting the depth of the tree drastically changes memory footprint and speed of the algorithms. My engine uses a somewhat high depth to ensure performance at the cost of significant memory, as the engine is expected to run on a game development machine. Other types of intersections may benefit from a lower depth, such as a frustum vs terrain test, which has many more node hits than the ray.

Because of this, I added a maximum depth parameter to some tests so that any granularity of nodes can be used.



GPU Brush Architecture

The central function of the terrain editor is sculpting, which means changing the topography of the landscape. In code, this means editing the heightfield image such that the terrain model is adjusted accordingly. There are some complications, however. Firstly, the image has to be updated efficiently so that it can be edited in real time. Second, the height data has to exist on both the GPU for rendering and on the CPU for efficient queries and to update the quad tree. Third, the edit function has to provide an interface for its size, location, and strength that may also be used for other tasks such as texture painting. The GPU brush architecture solves these problems.

The base class for brushes defines the interface below. Paint() is the central method that defines how the brush "paints" the terrain. Each brush also has a ShiftPaint() method that does some alternate operation, usually the inverse of Paint(). Each brush also receives update calls while it is active for updating internal state and the GUI.

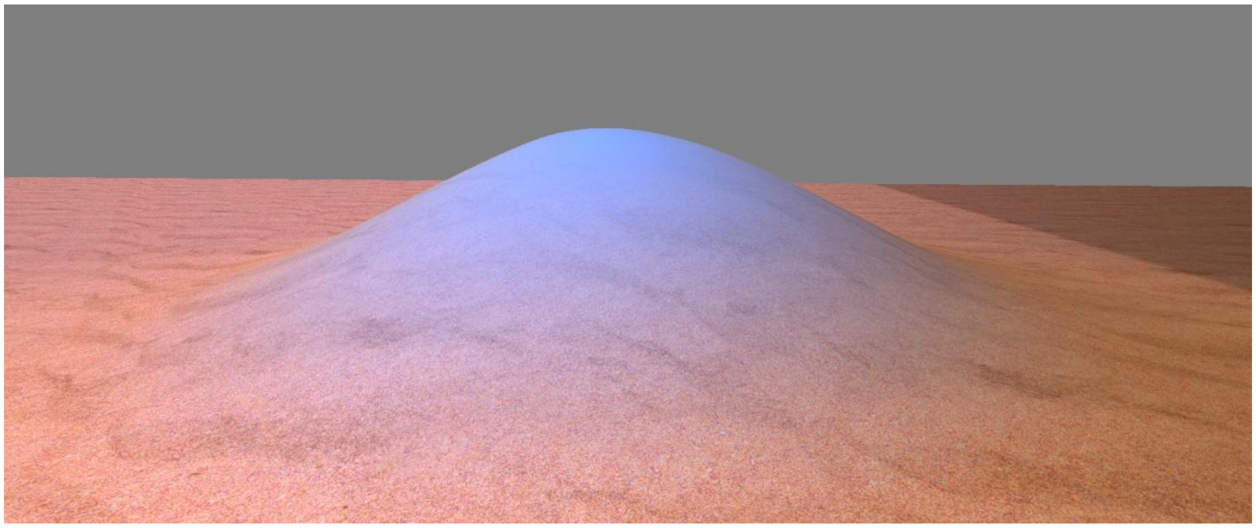
```
class ComputeShaderBrush
{
public:
    virtual PaintResult Paint(float radius, const Vec3& center, float force) = 0;
    virtual PaintResult ShiftPaint(float radius, const Vec3& center, float force) = 0;
    virtual void Update() = 0;
    virtual void UpdateGui() = 0;
     // ...
};
```

The sculpting implementation of the brush defines painting as editing the heightfield with some compute shader. The brush internally binds the heightfield texture as a read-write shader resource. Further derivations provide the concrete shader implementation and any additional resources needed. The shader implementation is then dispatched in parallel on the GPU based on the given location, size, and force of the brush. The shader is able to update the texture in VRAM extremely fast, but the height data needs to be pulled back down to the CPU. This is done asynchronously by using the **DO_NOT_WAIT** flag when mapping the texture back to the CPU. Each frame, the map is requested and eventually succeeds when the texture is ready. Then, the data is stored, and the quad tree is updated to reflect the new height values.

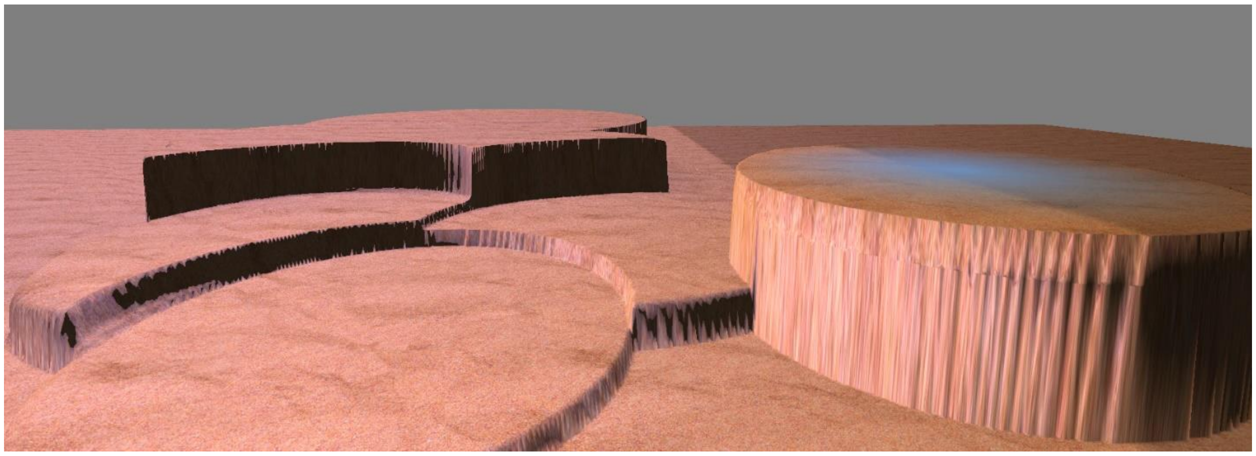
Updating the quad tree and other data on the CPU can be slow for large edits where millions of vertices may have changed. However, the data must be updated so that the user's mouse clicks continue to perform accurate intersections based on what they can see. To handle this, a maximum number of vertices in the changed area are permitted to be processed each frame after the download. The remaining vertices are queued to be processed in subsequent frames. This ensures a smooth minimum framerate at the cost of some latency in updating. This tradeoff is elegant because the latency only increases as the size of the edit becomes very large. So, for small, close-up edits, there is virtually no latency which allows for high user precision. On the other hand, for large edits, the camera tends to be very far from the terrain's surface, so the user is not as concerned with the exact intersection point of the mouse ray, so some latency in backend update is not noticeable.

Sculpt Modes

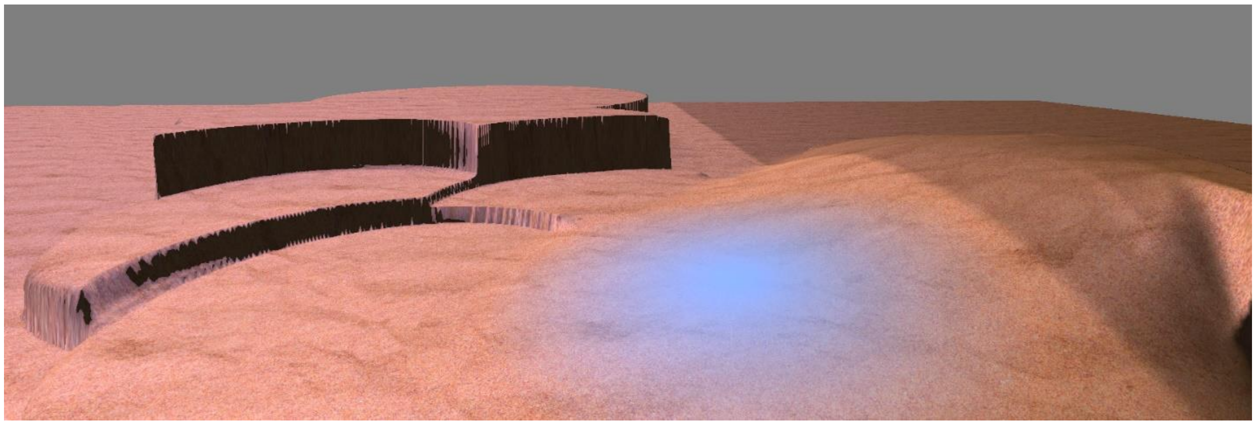
The most basic sculpt mode is the **Raise** brush. This simply raises and lowers the terrain creating mounds and valleys. In the shader, the thread ID is used to add the delta height to the heightfield image at the correct coordinate. The infrastructure needed to support this brush is reused for the more complicated implementations.

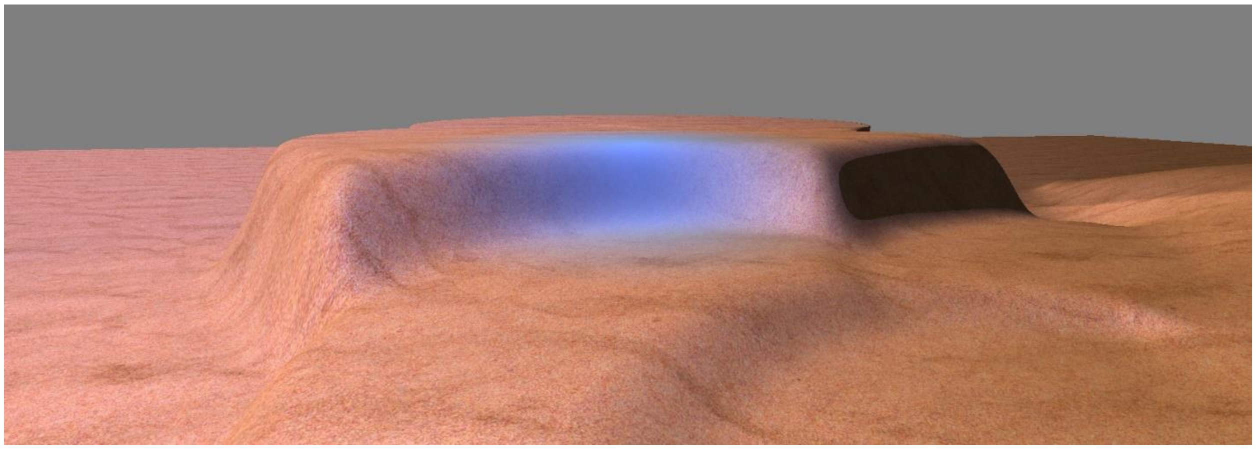


The **Flatten** brush flattens an entire area to a specific height. This is useful for creating plateaus or man-made areas. The target height can be sampled from the world using the alternate paint method.

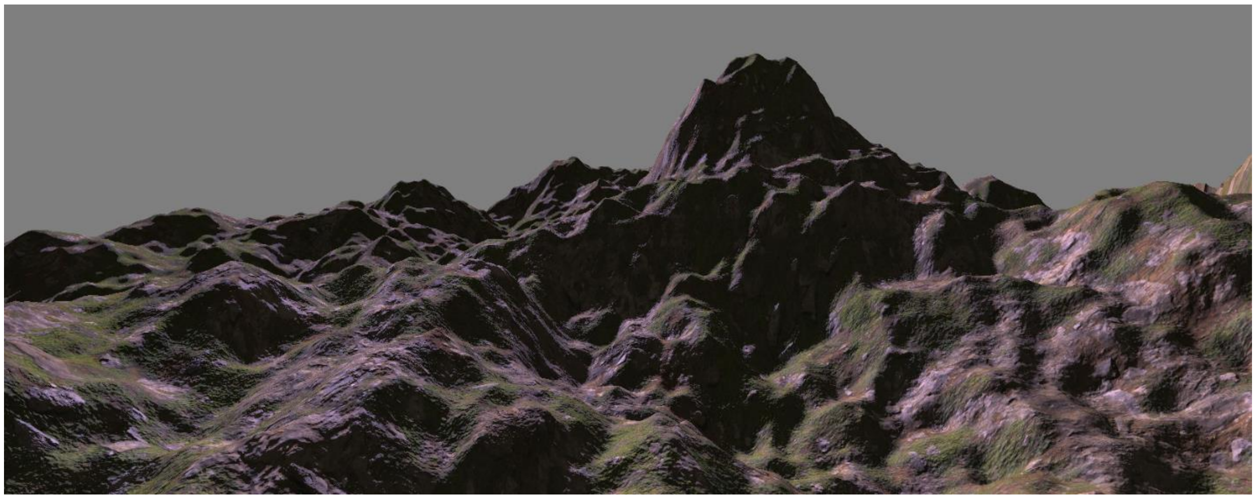


The **Smooth** and **Soften** brushes are two versions of a blur filter. The blur is applied to the terrain heightfield using a convolutional matrix, loaded as a buffer to the brush shader. Smoothing is an average blur while soften is a Gaussian blur. The average blur creates ramp-like features and completely averages out the area, while the Gaussian blur removes hard edges but preserves the existing character of the topography.





Lastly, I implemented a **Noise** brush which raises and lowers the terrain based on a noise texture. The brush can generate random Perlin Noise textures from the GUI which are bound to the brush shader as a resource. The texture is sampled and multiplied by the brush force to apply the noise to each vertex under the brush. This brush can be used to create roughness on the surface or even to generate entire mountain ranges. The alternate paint method subtracts from the heightfield, allowing a somewhat convincing erosion effect when an appropriate noise texture is used.



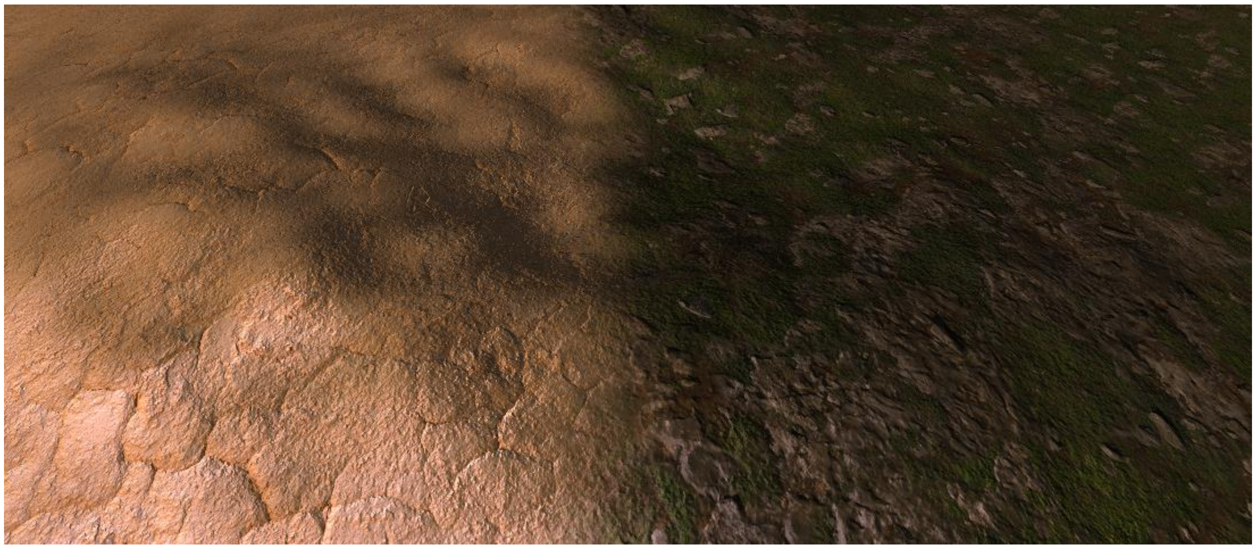
Texture Splatting

A good terrain system needs to support multiple textures in different areas of the landscape. These textures need to have smooth transitions between the different types. Also, the architecture needs to be conducive to editing the textures under the brush interface. To achieve these goals, my engine implements the texture splatting technique. This is a technique where a matrix of vectors is created. Each cell of the matrix represents a small area on the terrain, and the vector represents the weight of each available texture at that cell. The pixel shader then samples the weight vector based on the world position. For each weight in the sampled vector, the associated texture is sampled, multiplied by the weight, and added to the output. The resulting color is the weighted combination of each texture according to the vector.

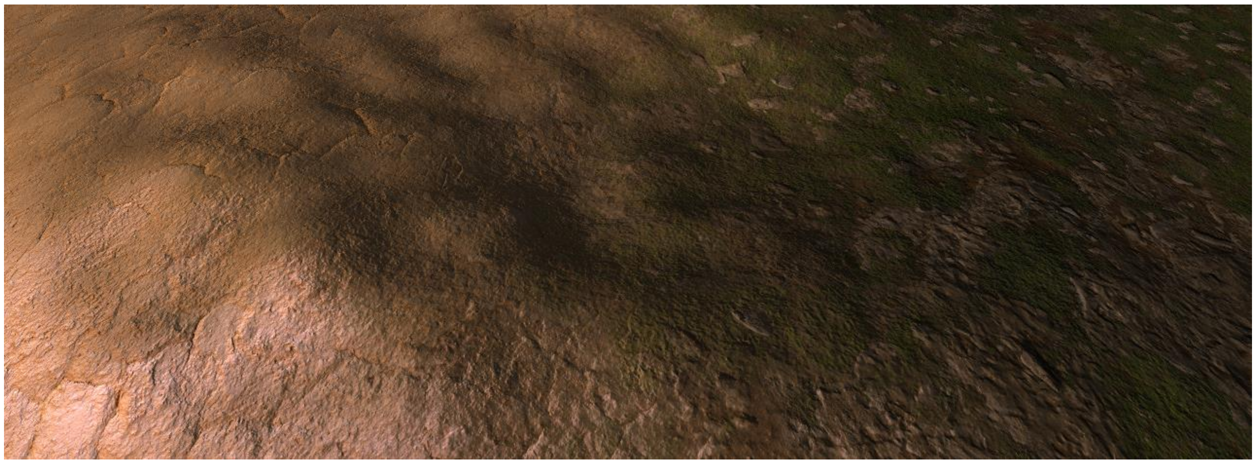
Texture splatting works well because it provides smooth, per-pixel blending between the textures. Also, the matrix of vectors is very similar to the heightfield, so we can create GPU brushes under the same interface that write to the texture matrix instead of the heightfield. Like the heightfield, the texture data needs to be downloaded from the GPU. This is mainly to be able to save the texture data to a file, but a game may also want to query the texture at a given location for various effects, such as the sound of footsteps.

Texture Paint Modes

The central paint mode is called **Paint**. This brush writes to the texture splatting matrix instead of the heightfield, as described previously. It sends an additional parameter to its compute shader, which is a layer mask vector. This vector tells the shader which element of the target vector to change without any conditionals. The layer mask is adjusted based on which texture the user wants to paint with.



The other paint modes are **Smudge** and **Gaussian Blur**. Smudging is an average blur filter that dramatically mixes all nearby textures together. The Gaussian Blur is a more subtle brush that smooths out any harsh transitions between texture zones. These brushes are very similar to their sculpting counterparts.



Foliage Architecture

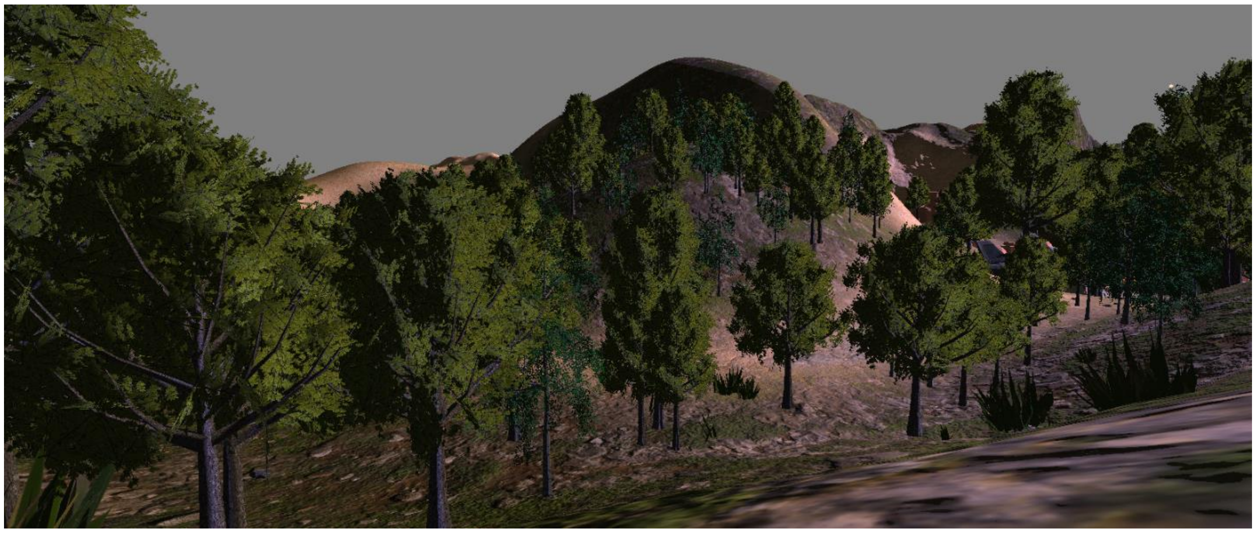
Foliage describes models added to the landscape as decoration, such as trees, grass, and rocks. Foliage presents unique challenges because a realistic scene needs to support an extremely high number of foliage instances on the terrain. Additionally, the engine needs to support painting and erasing foliage instances under the brush interface.

Rendering thousands of foliage models with a standard draw call is too slow. The go-to solution is GPU instancing, where each type of foliage is rendered with a single draw call with additional input data describing each instance. GPU instancing alone is not sufficient, because most of the foliage on the terrain is probably not visible or may be so far away that it does not need to be rendered. Hence, frustum culling is used to only render the instances that are necessary, allowing for much more dense foliage across the entire game world.

These optimizations affect how foliage information is stored. Firstly, each foliage type only needs one copy of its graphics resources. For each instance of that type, the world transforms and other flyweight properties are stored. These instances are organized into buckets based on the partition of the terrain they are contained in. This allows the rendering system to perform a camera frustum vs terrain test, using the quad tree, and only render the foliage in the visible partitions of the terrain.

Many vegetation models are simple quads that are mostly transparent. This requires some sort of blending to give the foliage the correct appearance. DirectX provides the **CD3D11_BLEND_DESC::AlphaToCoverageEnable** flag which magically blends everything correctly and is recommended for dense foliage.

The foliage lies on the surface of the terrain, but with the sculpt brushes, the topography might change. Therefore, each foliage instance samples the terrain height in the vertex shader to determine its world position, which allows the foliage to automatically update as the terrain is sculpted. Furthermore, some types of vegetation should follow the normal of the terrain, while others always stand roughly upright, like trees. So, a flag determines whether the normal is also sampled and used as the up axis.



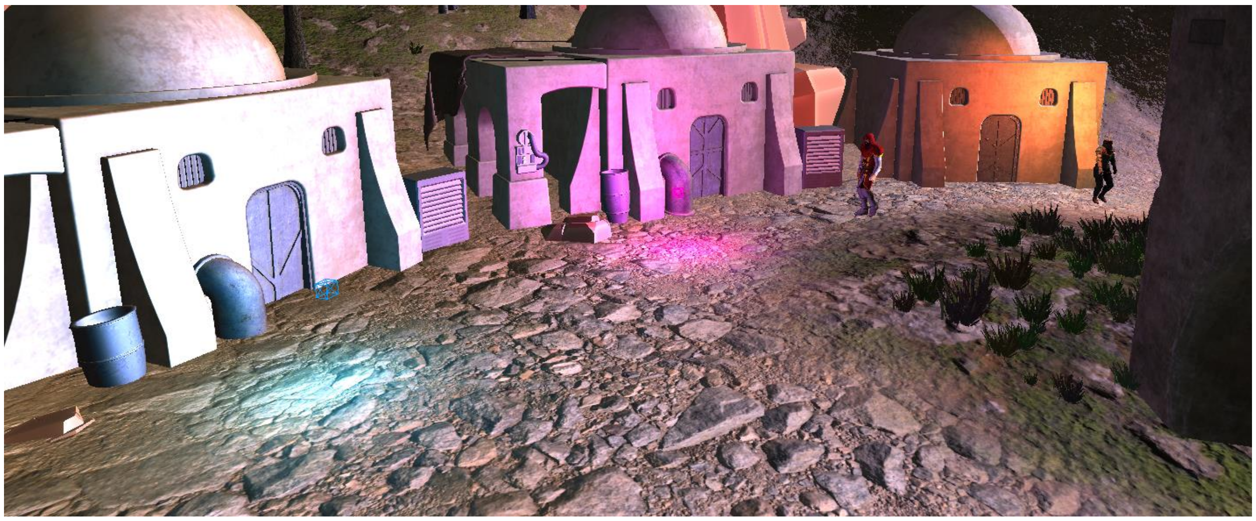
Foliage Brush

The **Foliage** brush extends the brush interface from sculpting and texture painting into the foliage system. This brush is not implemented as a shader, but rather uses the brush parameters to spawn instances under the brush's area. These instances are then rendered using the foliage architecture described previously. The alternate paint mode simply removes instances under the brush. The brush strength determines how quickly instances are spawned or removed, which is accomplished internally with a timer. The brush has additional parameters for random variations in rotation, scale, and color tint.

Lighting

To enhance the atmospheric world building capabilities of the engine and demonstrate the lighting effects of the landscape, I implemented a lighting system. The lighting system centralizes light data for each light type, which is referenced by each lit shader to create a consistent lit environment across all such shaders.

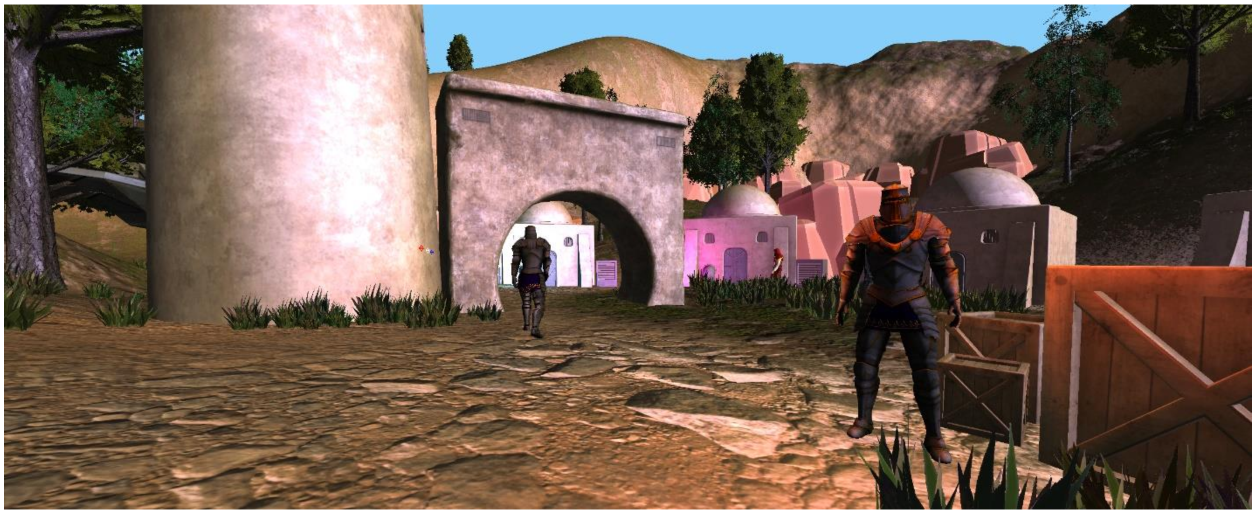
The light data is written to by light components that are attached to game objects. The light component contains all of the light properties for the given type. The game object transform informs some of the light properties referenced by the shaders. For example, a directional light component's direction property will be derived from the local forward axis of the light object. For point lights, the world position is used as the light position. This allows existing object transform widgets to be used elegantly to adjust the lighting in the scene and is similar to existing commercial engines.



The terrain is a lit model, and it implements some special techniques to improve the surface appearance. Firstly, the normals are calculated based on the average face normal around each vertex. The normals are calculated at runtime for simplicity, but a better system might store and update these in a texture similar to the heightfield. Additionally, the textures used in the texture splatting system can provide a normal map. This is sampled, based on texture weight, in the pixel shader to add per-pixel normals to the terrain surface. This greatly increases the visual quality of the terrain at minimal cost, and nicely fits into the texture splatting system.

Adventure Demo

To demonstrate the usability of the terrain and world editing system, I created a small adventure game demo. This demo imitates a modern top-down game where an animated, lit character can be moved across the terrain surface. The demo highlights how the sculpted terrain can be accurately queried to interpolate the height of the character using barycentric coordinates. Also, it shows the visual quality of texture layers with normal maps as the light moves. Dense and diverse foliage objects have been painted onto the landscape, and they are rendered efficiently. Lastly, it shows how some engine tools such as the mouse raycast can be exposed as an API to the game code for tasks like clicking on the terrain to control the character.



Improvements

One improvement to explore would be using a vertex and pixel shader to edit the heightfield for the GPU brushes. Here, a quad mesh could be positioned, scaled, and rotated based on the brush parameters. Then, a pixel shader would perform the texture edit. The upside would be the ability to rotate the quad, which is not possible with the current implementation. However, the simplicity of the compute shaders may be preferable to setting up a more complicated rendering pipeline.

Level of detail optimizations could be implemented in several domains across this system. Reducing the number of triangles in distant parts of the terrain and foliage would allow for an even larger scene.

One main limitation of this system is that the terrain size is fixed for any given terrain asset. It may be desirable to reimplement the system to support an “infinite” terrain to create a much larger world. This would require breaking up the terrain into pages, which could have additional improvements to data transfer speed between the CPU and GPU. The pages could then be loaded and unloaded dynamically depending on where the player or editor is looking.

Sources

ImGui
https://github.com/ocornut/imgui

Interactive GPU-based Procedural Heightfield Brushes
https://www.researchgate.net/publication/220795036_Interactive_GPU-based_procedural_heightfield_brushes

Effective GPU-based Synthesis and Editing of Realistic Heightfields
https://www.decarpentier.nl/downloads/EffectiveGPUBasedSynthesisAndEditingOfRealisticHeightfields_thesis.pdf

Fast Minimum Storage Ray-Triangle Intersection
https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

An Efficient and Robust Ray–Box Intersection Algorithm
https://people.computing.clemson.edu/~dhouse/courses/405/papers/bounding-box-tests.pdf

Efficient Intersection of Terrain Geometry in Real-Time Applications
https://odr.chalmers.se/server/api/core/bitstreams/a48948b6-2d80-49c2-ba68-cbc7e0895f5e/content

Perlin Noise
https://github.com/stegu/perlin-noise/tree/master

Instancing (With Indexed Primitives)
https://www.braynzarsoft.net/viewtutorial/q16390-33-instancing-with-indexed-primitives