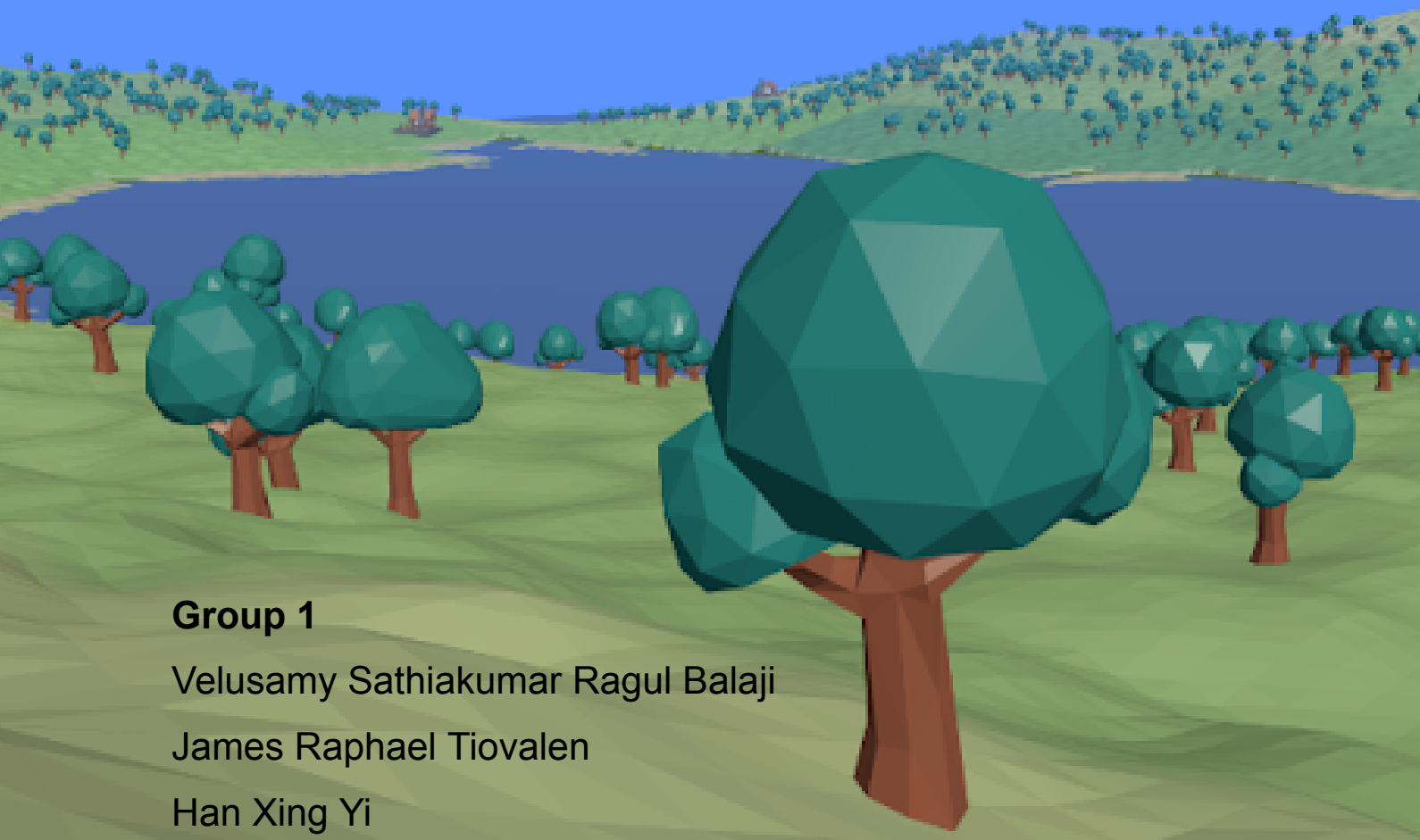
SOMAPAH WORLDSCAPES



Group 1

Velusamy Sathiakumar Ragul Balaji

James Raphael Tiovalen

Han Xing Yi

Qiao Yingjie

Huang He

Somapah Worldscapes

Procedurally Generated World by Design

1. Introduction

In our project, we applied various computer graphics techniques to develop a procedural & customisable world-generation sandbox. We were inspired by procedurally generated open-world games such as *Minecraft* and *Terraria* which have immense replay value as each world presents new terrain and challenges for players to explore and build in. As such, we wanted to study procedural terrain generation as it is an interesting and challenging fusion of many computer graphics problems such as noise layering, mesh generation, material shading & lighting, raycasting and instancing, etc.

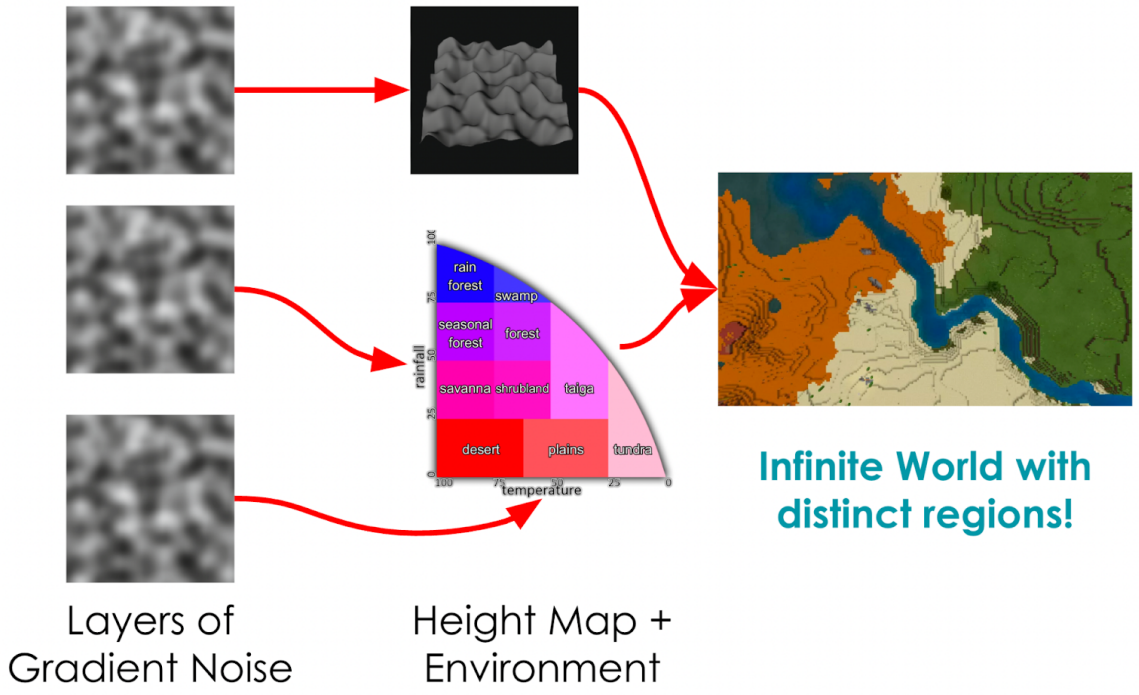
Our procedural terrain generation project is hosted as an interactive game which can be played within most **modern** web browsers (Google Chrome preferred). The player can generate different terrain maps by tuning a list of diverse parameters using our debug menu and exploring the map by flying around with their WASD & QE keys. First-person survival and creative mode can be toggled by pressing the C key.

You can play it here → http://ragulbalaji.com/worldscapes

2. Approach

2.1. Layers of Gradient Noise and Height Map

We aim to create an infinite world with distinct regions. To create terrains with diverse biomes, our approach is to heavily layer gradient noise maps at different resolutions (similar to old versions of Minecraft from 2009 & 2010). With multiple layers of gradient noises, we can create height, temperature and humidity maps which can then be interpolated to form different natural-looking formations, environments, biomes, flora and fauna.



For example, we look up the texture map in our shader by using the vertex's height as well as its associated temperature and humidity to derive its fragment shading information. Using this we can colour the mesh with the correct material colour before applying Phong shading, fog, and other lighting effects.

In this process, our custom shaders and lightning parameters in the GPU are deeply integrated with the noise generation done on the CPU to render the terrains in an efficient and visually appealing way. As the nature of such artistic optimization suggests, there are various tuning efforts for the noise map layering, shading, and materials. More details can be found in the Implementation section - Custom Phong Lighting Shader.

2.2. Chunk Mesh Loading

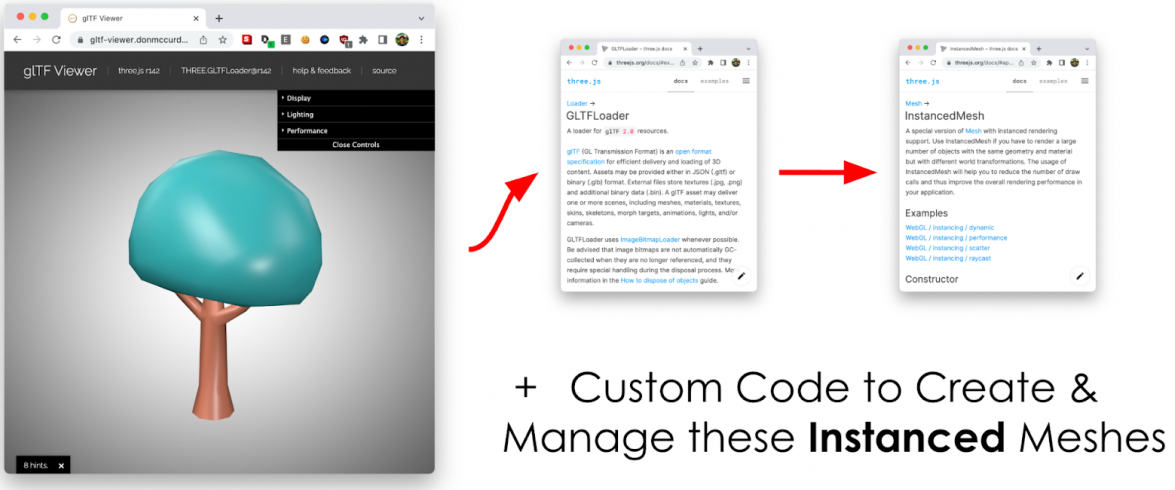
The basic unit of the terrain is a chunk. We need chunks as we are trying to sample and render an infinite amount of terrain on computers with finite RAM to store the vertex and shading information. Our approach is to load and unload chunks based on their relevance to rendering the current scene. To do this efficiently and maintain a good FPS, we ONLY perform the expensive makeChunk(...) generation function

ONCE when the player gets within render distance and the chunk is required to be loaded into the scene. This chunk and its associated decorations are stored on a hashmap until the chunk leaves render distances and our chunk garbage collector unloads it and frees its resources. To achieve the “illusion” of infinite terrain, we keep an N-by-N buffer of chunks loaded around the player, where N is a tunable and modifiable parameter. This yields stable and high FPS while keeping memory usage very low, leading to a good user experience when playing our game.

We can achieve continuous, smooth terrain between chunks without any seams as long as the noise functions used are continuous. This is due to the continuity properties of continuous functions, whereby the sum, difference, constant multiple, product, quotient, and even composition of continuous functions would be continuous.

2.3. Instanced Meshing

Instanced meshing is employed for efficient model object loading. Various GLTF decoration objects need to be loaded and added to the game scene such as the trees, houses, and wells. The issue is that the native GLTF loader is slow and expensive in terms of memory considering that we need tens of thousands of these static meshes for interesting terrain. Therefore, we convert all these GLTF objects to an array of instanced mesh objects, which gives us an extremely small memory footprint while also allowing us to set unique translation and rotation matrices per instance.



```
const loader = new THREE.GLTFLoader()
function loadInstancesOf (idx, GLTFpath, count) {
  loader.load(GLTFpath,
    function (gltf) {
      ALL_INSTANCED_MODELS[idx] = []
      gltf.scene.traverse(function (child) {
        if (child.isMesh) {
          const mat = child.material
          mat.flatShading = true
          const instancedMesh = new THREE.InstancedMesh(child.geometry, mat, count)
          instancedMesh.scale.set(8, 8, 8)
          instancedMesh.castShadow = true
          instancedMesh.receiveShadow = true
          scene.add(instancedMesh)
          ALL_INSTANCED_MODELS[idx].push(instancedMesh)
        }
      })
      loadedAssets++
    },
    function (xhr) {
      console.log((xhr.loaded / xhr.total * 100) + '% loaded --> ' + GLTFpath)
    },
    function (error) {
      console.log('GLTFLoader error ' + error)
    }
  )
}

const MODELS = {
  TREEA: [0, 'assets/models/gltf/detail_treeA.gltf.glb', 15000],
  TREEB: [1, 'assets/models/gltf/detail_treeB.gltf.glb', 15000],
  TREEC: [2, 'assets/models/gltf/detail_treeC.gltf.glb', 15000],
  WELLS: [3, 'assets/models/gltf/well.gltf.glb', 200],
  HOUSE: [4, 'assets/models/gltf/house.gltf.glb', 200]
}

let loadedAssets = 0
const ALL_LOADED_COUNT = 4

for (const MODEL of Object.keys(MODELS)) {
  const toload = MODELS[MODEL]
  loadInstancesOf(...toload)
}
```

This instanced meshing approach significantly improved the performance of our project which can run smoothly with very low memory consumption and a stable 120 FPS on an M1 Pro 16” Macbook Pro. More details can be found in the Performance Analysis section of the report.

3. Implementation

3.1. Perlin and Simplex Noise

To generate randomized terrains, we decide to use Perlin Noise [1, 2] to create our height maps, based on which we can create the diverse biomes in our game. Since the Perlin and Simplex Noise is a well-studied topic, we have decided to import the existing library to integrate the noise value generation into our project to accelerate our progress.

More detailed implementations can be found in the `makeChunk` function in game.js, and some important calculations are attached as follows. As indicated by the following formula, much manual tuning was necessary to create aesthetically appealing terrain generation and there are many fixed coefficients in the noise value generation as a result.

```
noise.seed(PARAMETERS.world_seed)
```

…

```
const temp = (noise.simplex2((x0 + x) / PARAMETERS.temp_divisor, (z0 + y) /
PARAMETERS.temp_divisor) + 1) / 2
let rain = (noise.simplex2((x0 + x) / PARAMETERS.rain_divisor, (z0 + y) /
PARAMETERS.rain_divisor) + 1) / 2
```

…

```
let h = noise.simplex2((x0 + x) / 8, (z0 + y) / 8) * (rain + 0.3) +
        noise.simplex2((x0 + x) / 128, (z0 + y) / 128) * 4 +
        Math.min(32, noise.simplex2((x0 + x) / 768, (z0 + y) / 768) * 32 + 16)

const m = Math.abs(noise.perlin2((x0 + x) / PARAMETERS.mountain_divisor, (z0 + y) /
PARAMETERS.mountain_divisor))
const mt = Math.abs(noise.simplex2((x0 + x) / PARAMETERS.mountain_divisor, (z0 + y) /
PARAMETERS.mountain_divisor) * 0.2)
```

…

```
const troty = noise.perlin2((x0 + x) * PARAMETERS.troty_multiplier, (z0 + y) *
PARAMETERS.troty_multiplier) * Math.PI * 2
```

3.2. Custom Phong Lighting Shader

We decided to follow a hybrid approach when implementing our custom shader. The built-in shaders have useful primitives that we would prefer to keep, but at the same time, the built-in shaders are insufficient for our use case. As such, we “hijack” the compilation of the built-in shader and “inject” our own shader code to allow us to perform what we need. This was done by defining the onBeforeCompile callback method of the material that we are using to create our mesh with our own vertex shader and fragment shader code that we would like to inject. Some pseudocode would be as follows:

```
const phongMaterial = new THREE.MeshPhongMaterial({
  // Define some material parameters here
})

phongMaterial.onBeforeCompile = function (materialInfo) {
  // Customise the vertex shader
  materialInfo.vertexShader = ...
  // Customise the fragment shader
  materialInfo.fragmentShader = ...
}
```

This allows us to generate the terrain meshes using our noise-based method, while at the same time, applying Phong lighting to the terrain without completely rewriting any built-in shaders.

For our specific implementation, we had to analyze the GLSL source code of the shader used by MeshPhongMaterial in three.js in order to identify the corresponding sections that could be overwritten, replaced, or modified. Looking through the shader file here, it seems that the descriptions of both the vertex shader and the fragment shader are split into chunks, which are accessible here. Browsing through the different files, we can then slowly construct and build the shader that we want.

We first set vertexUvs to true and uvsVertexOnly to false. This is because, based on the parameter mappings specified in this file, these flags would define, and thus enable the USE_UV preprocessor directive and disable the UVS_VERTEX_ONLY preprocessor directive respectively. Hence, we would effectively have access to the following shader line of code:

```
varying vec2 vUv;
```

Then, we insert the following lines of code into the vertex shader:

```
varying vec3 vPosition;
```

…

```
vPosition = position;
```

We also insert the following line of code into the fragment shader:

```
varying vec3 vPosition;
```

Another point of interest to us that is present in the fragment shader is this file. By passing in a texture colour map parameter into the material, we would have access to the texture map via the following line of code:

```
uniform sampler2D map;
```

Since we want to inject some custom fragment shader code, we decided to overwrite the following original piece of code from the <map_fragment> section:

```
vec4 sampledDiffuseColor = texture2D( map, vUv );
diffuseColor *= sampledDiffuseColor;
```

Instead, we replaced the code above with the following code:

```
if (vUv.x > 0.8 && vUv.y > 0.8) {
  vec4 sampledDiffuseColor = vec4(0.1, 0.2, 0.7 - vPosition.y, 1.0);
  diffuseColor *= sampledDiffuseColor;
} else {
  vec4 sampledDiffuseColor = texture2D( map, vec2(vUv.x, vPosition.y / 64.0) );
  diffuseColor *= sampledDiffuseColor;
}
```

This allows us to shape the terrain mesh and perform texture mapping using the input texture map accordingly, depending on the noise map values, all done efficiently in the GPU!

3.3. Camera Control

In our worldscapes, the camera is controlled like a first-person shooter (FPS), where the WASD keys or arrow keys on the keyboard are used to control the player's movement and the cursor is used to control the player's viewing direction. We implemented this FPS control from the ground up, using two vectors with 3 components (x, y, z) to keep track of the position of the player (camera) and the velocity of the player; we monitor the key presses at all times and update their status in an object map; and based on this object map, we compute the change in velocity and displacement for each delta time between two rendered frames.

In free-flying (i.e., creative) mode, we treat the current viewing direction as the forward direction for player movement, which enables the player to move at any arbitrary angle instead of just along the x, y, and z axes, we use two helper functions to get the normalized forward and side vector respectively, and multiply it with the speed constant to obtain the change in velocity.

```
function getForwardVector () {
  camera.getWorldDirection(playerDirection)
  playerDirection.y = 0
  playerDirection.normalize()
  return playerDirection
}

function getSideVector () {
  camera.getWorldDirection(playerDirection)
  playerDirection.y = 0
  playerDirection.normalize()
  playerDirection.cross(camera.up)
  return playerDirection
}
```

We allow the player to use the cursor movements to change the current viewing direction. We use Euler angles with a “YXZ” rotation order to achieve camera rotation because Y rotation (yaw left and right) is much more frequently used than X

rotation (pitch up and down) and thus we allow Y to rotate freely without hitting a gimbal lock. We first acquire the quaternion from the camera and convert it to Euler Angle. Then, we update the x and y rotation based on the cursor displacement on the screen in the opposite direction. We further set a hard limit to the X rotation to be within [-90, 90] degrees to avoid gimbal lock before converting it back to quaternion and set the new viewing angle for the new frame.

```
function onMouseMovement (e) {
  if (!pointerLocked) return
  const movementX = e.movementX || e.mozMovementX || e.webkitMovementX || 0
  const movementY = e.movementY || e.mozMovementY || e.webkitMovementY || 0
  eulerAngle.setFromQuaternion(camera.quaternion)
  eulerAngle.y -= movementX * 0.002 * POINTER_SPEED
  eulerAngle.x -= movementY * 0.002 * POINTER_SPEED
  const minPolarAngle = 0
  const maxPolarAngle = Math.PI
  eulerAngle.x = Math.max(
    _PI_2 - maxPolarAngle,
    Math.min(_PI_2 - minPolarAngle, eulerAngle.x)
  )
  camera.quaternion.setFromEuler(eulerAngle)
}
```

3.4. Gravity and Collision Detection

To create a realistic “walking” experience for the player in our Somapah Worldscapes, we implement simple physics like gravity, air resistance and player-terrain collision. Gravity and air resistance are implemented by applying a constant acceleration with a small amount of decay in the vertical axis when the player is not “on the ground”.

To determine if the player is “on the ground”, we need to detect collisions between the player and the terrain. As we don’t need to deal with collisions with overhanging elements such as roofs, we simplify the collision detection implementation by using a single raycaster that always originates from the camera position and points down. If the ray intersects with any mesh faces, we get a distance value from the raycaster and the normal vector of the face that the ray is intersecting. When the distance to the first intersecting element is below a certain threshold, we declare a collision and we apply a velocity to the player in the reverse direction of the face normal. To

further optimize the performance of ray casting, instead of having the ray intersect the entire currently generated terrain mesh, we select the chunk right below the player (camera) position and have the ray cast only on this chunk. This improves the raycasting performance by a factor of N, where N is the number of chunks loaded.

```
function playerCollisions () {
  if (CREATIVE_MODE) return
  // set ray origin to player position and points downwards
  rayCaster.set(playerPosition, new THREE.Vector3(0, -1, 0))
  // select the chunk right under the player
  const chunkX = Math.floor(camera.position.x / PARAMETERS.chunk_size + 0.5)
  const chunkZ = Math.floor(camera.position.z / PARAMETERS.chunk_size + 0.5)
  const chunkName = `${chunkX}$$${chunkZ}`
  // ray cast against the chunk
  const intersects = rayCaster.intersectObjects([loadedChunks.get(chunkName)])
  // if there is an intersection
  if (intersects.length > 0) {
    // if the distance to the intersection is less than the threshold
    if (intersects[0].distance < PLAYER_HEIGHT) {
      const normal = intersects[0].face.normal
      // set player is on the floor
      playerOnFloor = true
      // apply a force in the opposite direction of the normal
      playerVelocity.addScaledVector(normal, -normal.dot(playerVelocity))
    } else {
      playerOnFloor = false
    }
  }
}
```

3.5. Memory Management

To avoid exploding memory, we set a user-adjustable constant to limit the maximum number of chunks that can be loaded at any single time. Whenever the number of loaded chunks hit this threshold, we will try to unload 20 percent of the furthest chunks from the player's current position.

```
// Unload chunks
if (loadedChunks.size > PARAMETERS.max_num_chunks) {
  const chunkNames = Array.from(loadedChunks.keys())
  chunkNames.sort((a, b) => {
    const aDist = loadedChunks.get(a).position.distanceTo(playerPosition)
    const bDist = loadedChunks.get(b).position.distanceTo(playerPosition)
    return bDist - aDist
  })
```

```
  for (let i = 0; i < chunkNames.length - PARAMETERS.max_num_chunks * 0.8; i++) {
    const chunkName = chunkNames[i]
    unloadChunk(chunkName)
  }
}
```

Additionally, we also perform incremental rendering. In other words, we load all of the necessary chunks over multiple frames instead of loading all of those chunks in a single frame, allowing the CPU to spread out the intensity of the work involved in mesh generation and pushing buffers to the GPU. This might cause a slight, negligible scan-line effect whenever new chunks are loaded or whenever old chunks are unloaded.

3.6. Parameter Live Reloading

When the user modifies the parameters in the debug menu, the GUI would perform a live reload of the entire scene with the new values if necessary. This is because, while some parameters such as lighting configuration and material properties can be easily changed without the need of reloading all of the loaded chunks, other parameters such as the world seed, maximum number of loaded chunks, and noise parameters directly affect the mesh of the chunks themselves. As such, for these parameters, all of the loaded chunks are unloaded before being reloaded again using the new parameter values, which might cause a quick flicker/scan-line effect across the chunks. This allows for live reload capabilities without severely affecting the user experience.

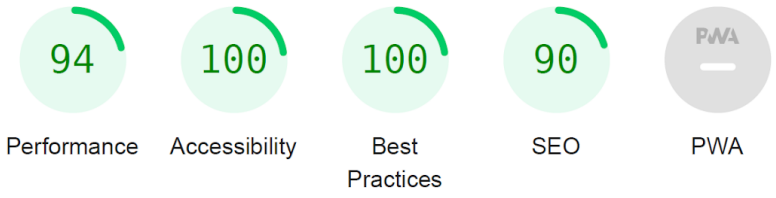
4. Results

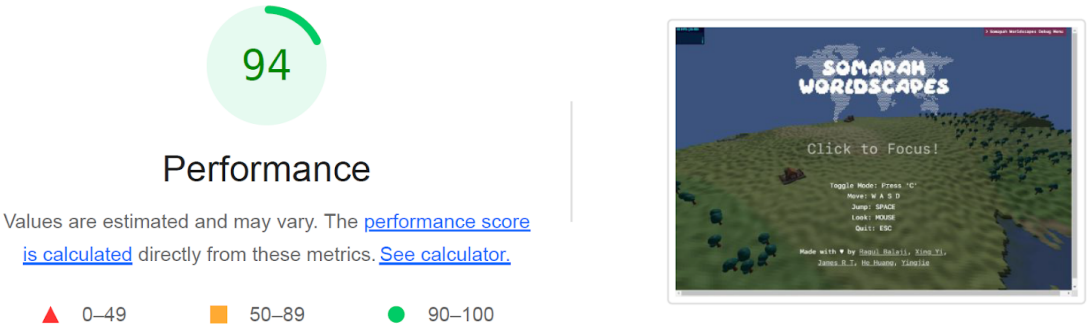
4.1. Demo Video

The demo video is available at https://youtu.be/Oug6xmuTke0
Additionally, the game is hosted live at http://ragulbalaji.com/worldscapes/.

4.2. Performance Analysis

Google Lighthouse Report:





As shown on the Google Lighthouse report above, our performance remains high and we can still achieve up to 120 FPS (limited by V-Sync) with less than 200 draw calls even when we render millions of vertices and meshes.

5. Discussion

5.1. Pros and Cons

Our project distinguishes itself in many aspects including an interesting and diverse landscape with various decoration objects for players to explore, and high

replayability as a game thanks to the random and procedural terrain generation. As a result, it naturally sparks creativity in players.

Some certain limitations and areas are worthy of further exploration. One technical difficulty we encountered is the manual tuning of random noise value functions, which is time-consuming and hard. Even after much tuning, the result might still not be perfect, for which we speculate a more advanced random function than Perlin and Simplex Noise is warranted. For the game itself, the objects are designed such that they are not destructible, and it is hard to store deltas.

A potentially better approach that could be explored in future work would be to integrate Voronoi diagrams to partition biomes during terrain generation. These Voronoi diagrams could then be combined with the noise functions to generate more realistic biome placements. Finally, other special landscapes could be explored, such as implementing caves using Perlin worms and terraces by clamping heights to discrete steps.

5.2. Learnings and Take-Aways

As a team, we have many learnings and takeaways as we work our way through the project. We managed to gain a much deeper understanding of many computer graphics concepts covered in this course as we utilize those concepts in this hands-on project including Mesh Modelling, Shaders for Texturing, Phong Shading Model, 3D modelling and re-meshing. Moreover, as we are developing the project to be shipped in the format of the game, we have also improved our understanding of various software engineering topics such as efficient Memory Management, Performance Optimization (in the context of a web browser), and UI/UX and interactivity.

6. Conclusion

In conclusion, we have successfully delivered our computer graphics project in the format of a game. Importantly, our efforts in the following topics are significant in

creating an effective procedural terrain generation and a highly performant game with efficient resource utilization:

- Using noise layers to generate terrain
- Using shaders to generate environments
- Using instance meshes to add trees & homes

And most importantly, as a game, we have successfully made it easily available via a web browser (http://ragulbalaji.com/worldscapes/) and had much fun in the process!

7. References

1. Procedural Generation: Perlin Noise: https://www.cs.umd.edu/class/fall2018/cmsc425/Lects/lect14-perlin.pdf
2. Detailed Explanation on Random Biome Generation Using Perlin Noise: https://gamedev.stackexchange.com/a/186197
3. Making Maps With Noise Functions: https://www.redblobgames.com/maps/terrain-from-noise/
4. Three.js Library: https://threejs.org/
5. lil-gui Library: https://lil-gui.georgealways.com/
6. KayKit Medieval Builder Pack: https://kaylousberg.itch.io/kaykit-medieval-builder-pack