Bomb Arena: Project report

FRI, University of Ljubljana

Subject: Računalniška grafika in tehnologija iger (RGTI)

Authors: Miha Bernik, Anže Pečaver

Date: 10. 12. 2023

Table of Contents

Introduction

This report beings by overviewing the design and gameplay aspects of "Bomb Arena," a cartoonish, two-player arcade game set in a stone-brick arena floating in space. Players, depicted as simplistic astronauts in red or blue, navigate the arena by placing bombs, with gameplay focusing on strategy and quick thinking due to the game's random generation of arena and power-ups. Following the gameplay overview, the report describes the technical details, focusing on graphical aspects, including the shadowmap pass, geometry pass, overlay pass, and the final composition pass. Sources used during development and for the assets used in the game are listed at the end of the report.

1. Game design and gameplay overview

Bomb Arena is a two-player arcade game where players compete to blow each other up using their bombs. It is suitable for all players, from beginner to advanced. The game is styled in a cartoonish manner as an arena made of stone bricks and dungeon type items, while floating in space. The players are depicted as simplistic astronauts and are given the distinct color of either red or blue. The user interface is simple, showing the current time and information about the players (bombs available, score, lives left). Particle effects are used for the explosions, as details and as power ups visible in the arena.

The camera is positioned above the players, follows their average position and zooms in and out depending on their distance, so that both players are clearly visible in the scene.

The arena is rectangular with partially randomly generated tiles, based on a preset generation map, some of which are destructible, and some are not. The players must navigate through this arena by blowing up the destructible tiles and avoiding being caught in the explosion.

The players begin with one bomb in their inventory. They may place it in the position that they are in. After a two second delay, the bomb explodes in the four directions in a “+” shape, stopping at the first indestructible tile, after the first destructible tile or at a certain bomb radius, initially at one tile away from the center. If a player is caught in the explosion, regardless if the bomb was placed by them, the player dies.

The players each have three lives and lose a life upon death. Following a lost life, the player respawns at their initial position after a three second delay and are granted a three second invincibility period, during which they cannot die. When the player has died three times, the game is over and the other player wins. However, if both die at the same time with no lives left, the game ends in a draw.

The game features power-ups, which appear in random empty tiles in the arena every five seconds, up to a maximum of ten power-ups being present at once. They are randomly selected upon generation from three types of power-ups, which increase the player’s movement speed, bomb radius and bomb capacity. These power ups serve to give a players an edge and motivate them to explore the arena, rather than follow the quickest path to the other player.

The game uses a simple battle-type background soundtrack. Sound effects are used for explosions, player deaths, power-up pickups. The sound effects are simple as they simply play an audio file upon an event – the game does not use range to calculate the volume of the sound effect, for example.

The result of the explosive nature of the game is fast-paced, quick-thinking but also long term strategic gameplay. The random generation of both the arena and the power-ups means no two matches are the same, which greatly increases the replay potential of the game.

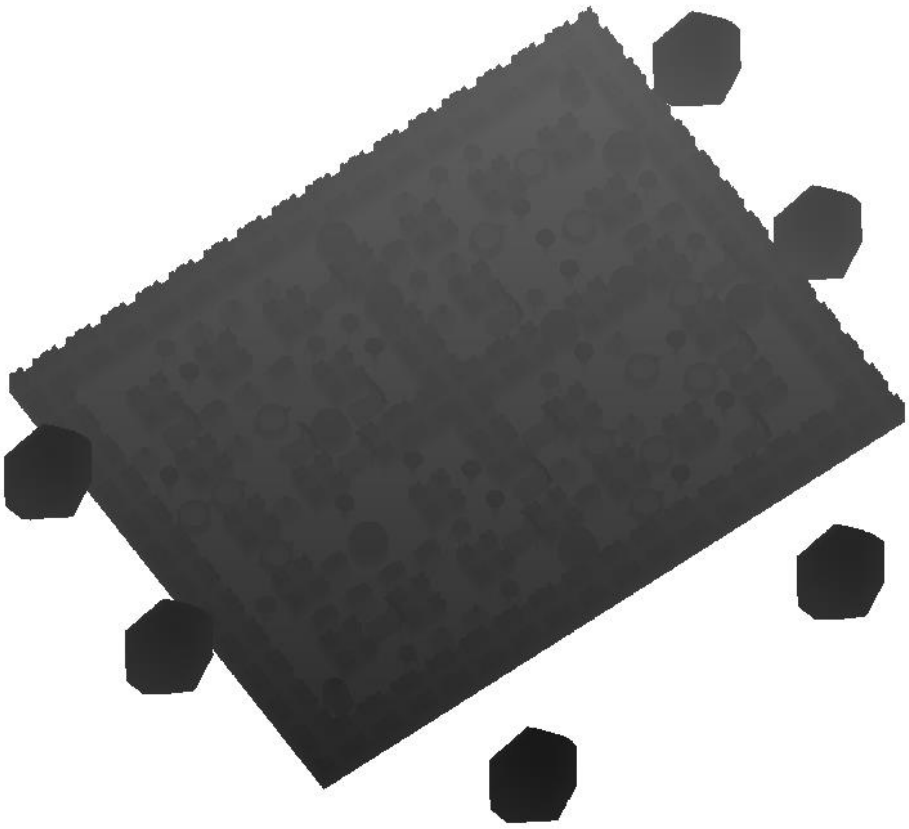
2. Graphical aspects

2.1 General information

To handle the extensive duplicate geometry, we employ instanced rendering for each mesh/material combination, resulting in approximately 10 geometry draw calls. When considering the overall process, shadow mapping effectively doubles these geometry draw calls. Additionally, there's an extra draw call for particle effects, text rendering, and the final fullscreen quad. The use of mipchains in textures significantly enhances the visual quality, particularly for normal maps. Furthermore, we minimize the data sent to the GPU by updating only the altered segments in the GPU buffers as necessary.

2.2 Shadowmap pass

The initial step in the process involves a shadowmap pass, where the geometry is rendered from the direction of the sun using an orthographic camera, as the light rays are nearly parallel. This pass generates a depth attachment with a resolution of 2048x2048, which is sufficient for our purposes. It's notable that only the vertex shader is invoked in this pass, as only depth information is required. An interesting aspect is that the rendering cost of the shadowmap, which excludes fragment shaders, is about 60% that of standard rendering involving fragment shaders, depending on scene complexity.



2.3 Geometry pass

The second step in the rendering process is the geometry pass. It renders all the geometry from the camera's perspective and includes simple phong lighting, normal mapping and applying the shadowmap.

Object transforms are stored in SSBO (Shader storage buffer object) and are indexed in the shader by instance id + given offset.

2.3.1 Phong lighting

Phong lighting is calculated as follows:

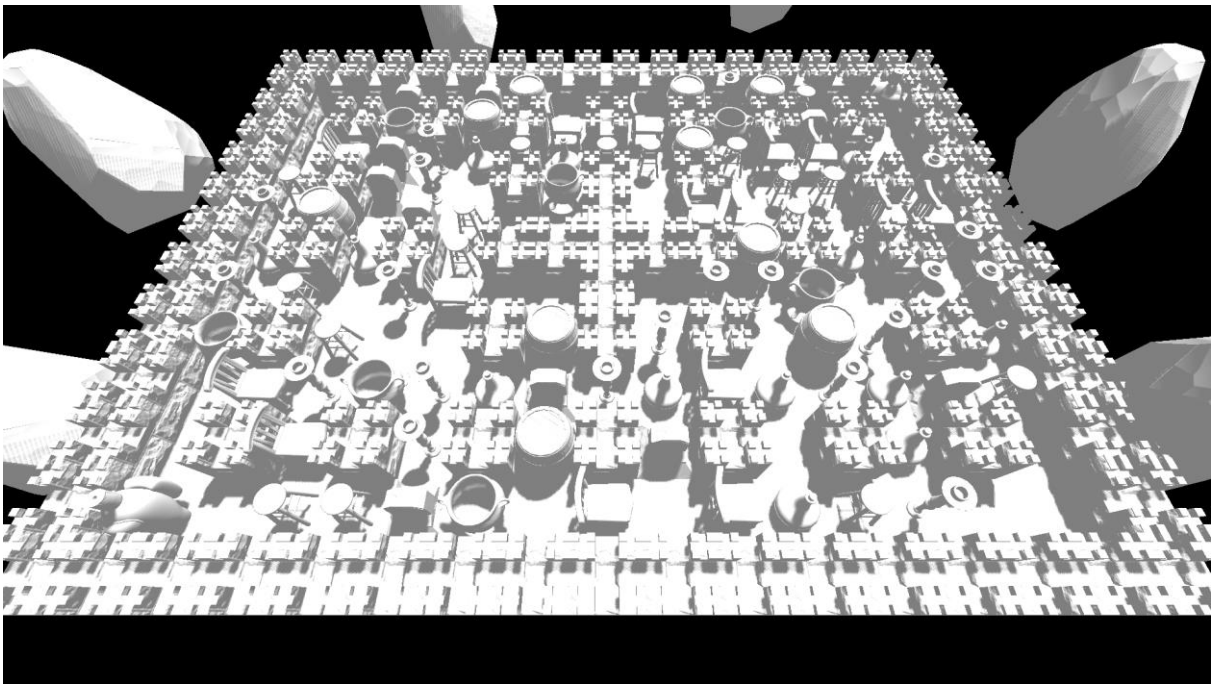
Simple ambient + diffuse + specular lighting

2.3.2 Normal mapping

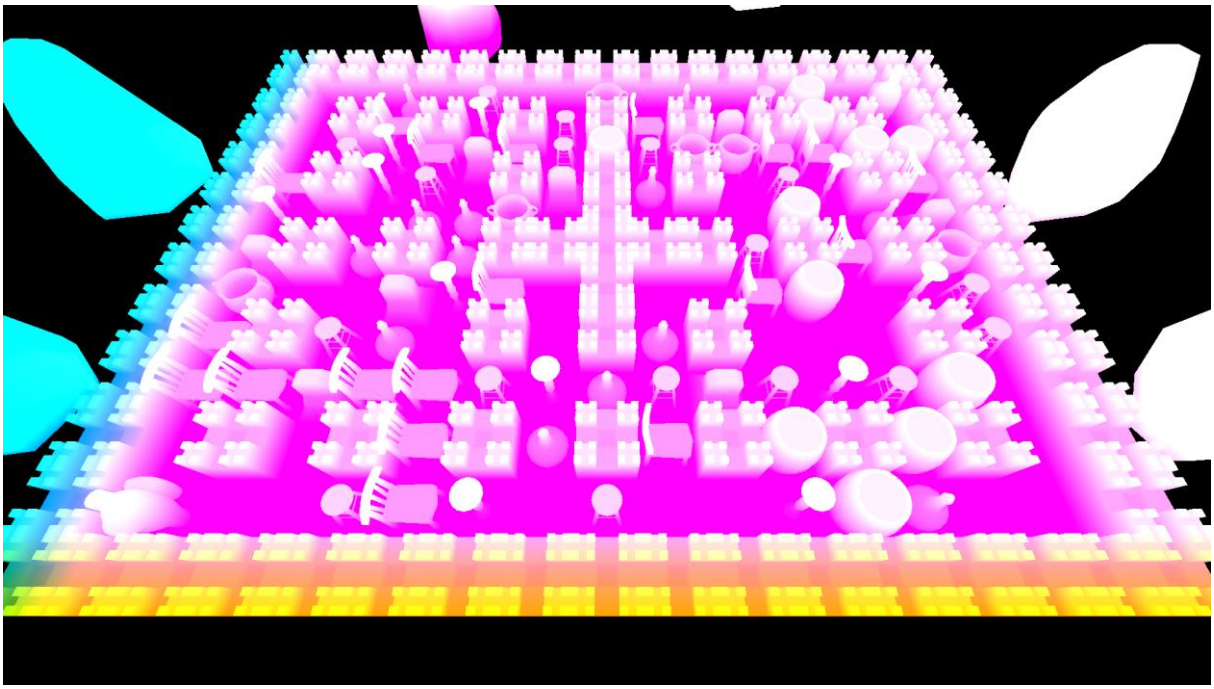
Normal maps transferred to the GPU are in tangent space. To translate them into world space, we require the Normal, Tangent, and Bitangent vectors within the shader. These vectors are orthonormal, meaning only two are needed to compute the third. We opt for the Normal, already being passed to the shader, and the Tangent, calculated during model loading. The Bitangent is derived as the cross product of the Normal and Tangent. A 3x3 TBN matrix (Tangent, Bitangent, Normal), created by concatenating these vectors, is used to determine the final normal direction, calculated as TBN×(sampled normal×2.0−1.0).

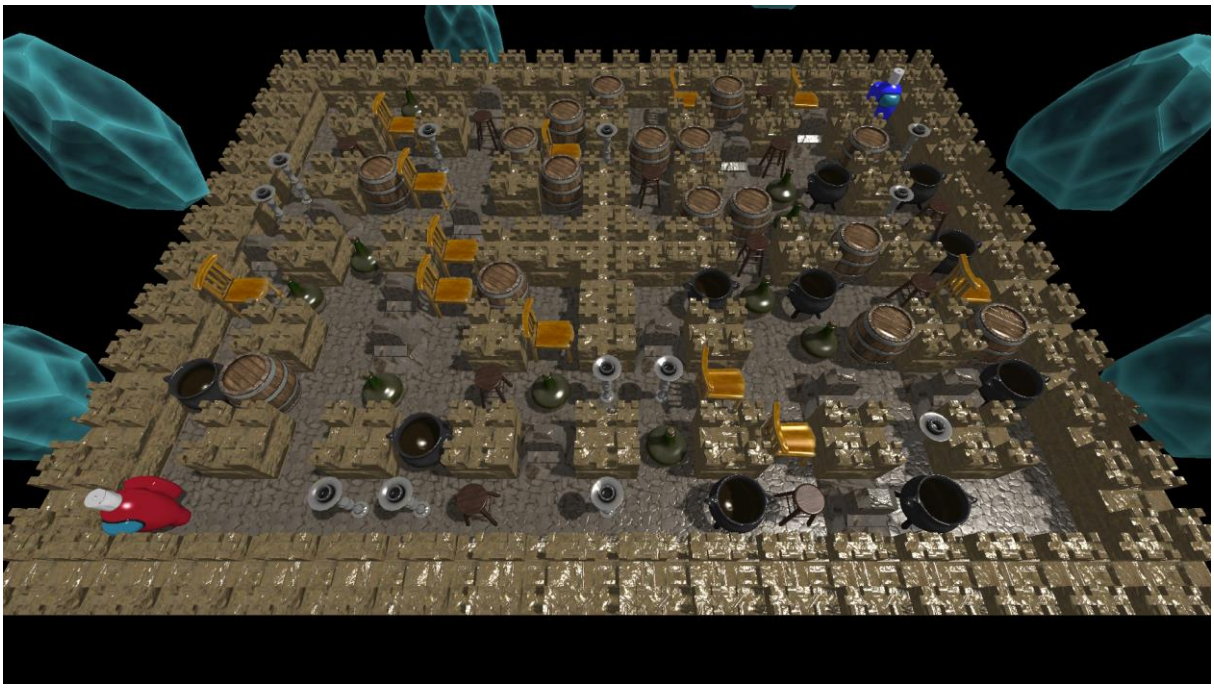
2.3.3 Shadowing

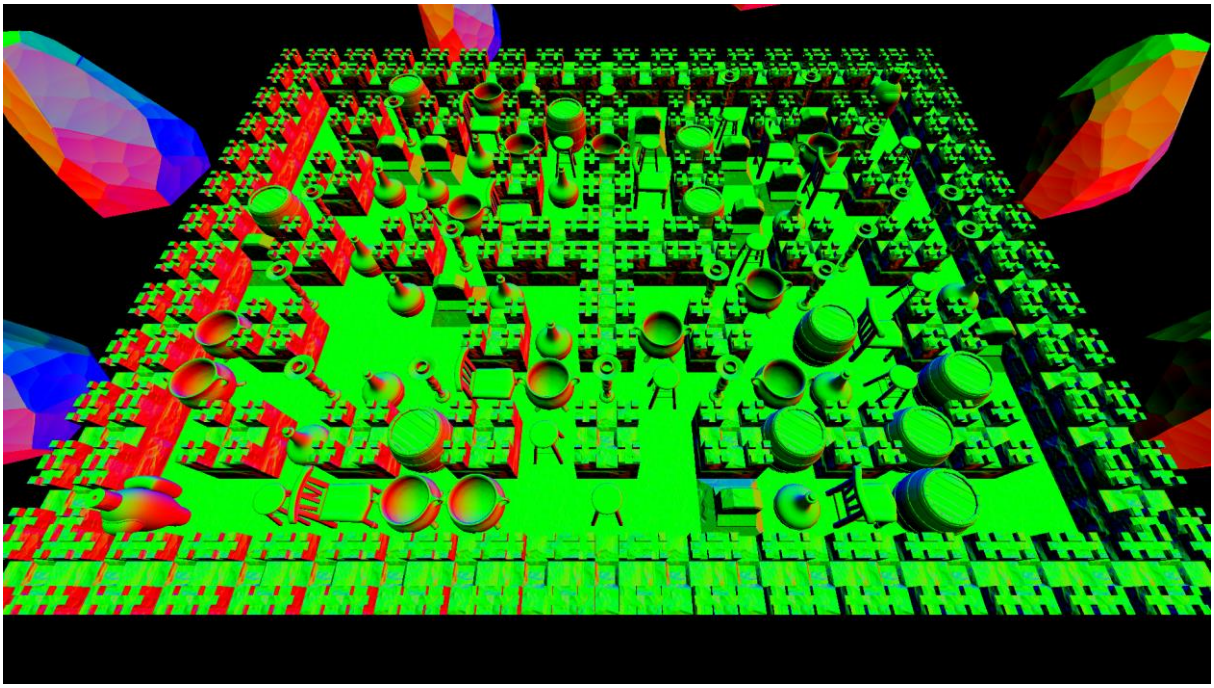
By utilizing the same view-projection matrix from the sun used in the shadowmap pass, we can determine the clip space coordinates and depth from the perspective of the sun for our camera. This information is then compared against a texture lookup from the shadowmap pass. Additionally, we implement Percentage-Closer Filtering (PCF) to create softer shadows. This is achieved by examining a 3x3 grid surrounding the sampled point.



The geometry pass in the rendering pipeline includes four attachments: position, color, normal, and depth.





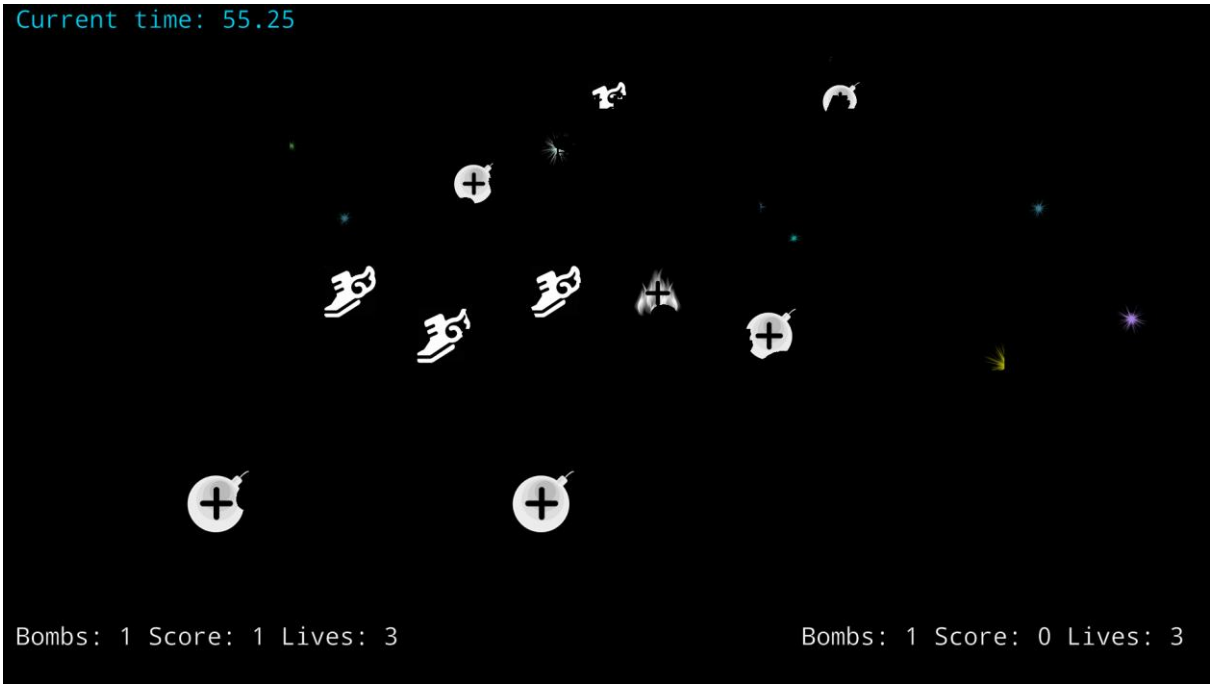


2.4 Overlay pass

The third step in our rendering process is the overlay pass, where we render particle effects and text onto a separate color attachment for use in the later composition pass. Our particle system utilizes instanced quads, which reference an SSBO containing data for up to 4096 particles (including position, rotation, color, radius, and texCoord). These quads, always facing the camera, are rendered according to their SSBO index. We use a large texture atlas for particle textures to minimize separate draw calls and resource bindings.

For text rendering, we use a technique called Multi-channel Signed Distance Field (MSDF). MSDF employs a texture that records distances to the edges of letters, using multiple channels for sharp corners. This method allows for almost infinite magnification of text without loss of quality, overcoming the limitations of traditional rasterized textures, which can degrade in resolution when viewed up close.

The MSDF technique, useful for rendering fonts, is also applicable to emojis, logos, and any vector-based art. In the overlay pass, text is batched into a single vertex buffer and updated on the GPU each frame. This pass utilizes the depth attachment from the geometry pass, enabling depth testing while disabling depth write to avoid blending artifacts and negating the need to pre-sort particles. The overlay pass outputs to a separate color attachment, distinct from the one in the geometry pass, to facilitate later Screen Space Reflection (SSR) implementation.



2.5 Final composition pass

The fourth and final step in our rendering process is the composition pass, where the results of all previous passes are combined into one final image. This image is then passed to the swapchain.

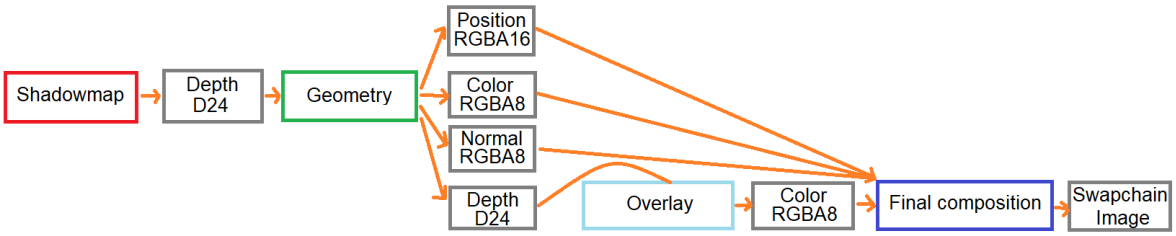
The background scrolling effect is achieved by offsetting the texture coordinates based on elapsed time. The sampler is set to repeat the texture when it extends beyond the [0, 1] range, which requires a seamless background image. Initially, a parallax effect similar to the Minecraft end portal was considered, but our final decision was to use simple texture scrolling.

Screen Space Reflection (SSR) is a post-processing effect that simulates reflections for only those objects visible on the screen. The process starts by determining the world space position of the fragment being processed by the GPU. We then compute a ray based on the normal at this point in world space, with both points converted into screen space. By marching along the ray and the line in screen space, we identify the position and UV coordinates of the traversed fragment. Comparing the distance to the initial point with the distance obtained by sampling the position attachment from the geometry pass helps in determining if the ray intersected with any object, allowing us to read the fragment's color.

However, this method has drawbacks, such as the potential to skip pixels along the ray, which is especially noticeable on smooth surfaces. The use of normal maps tends to skew reflections, effectively masking these imperfections. Ideally, the Digital Differential Analyzer (DDA) algorithm, which visits every pixel once along a line, would be more efficient. Additionally, using the depth attachment instead of the position attachment could significantly improve memory usage, as one can be computed from the other.

The challenge in implementing the DDA algorithm is in WGSL's requirement for uniform control flow, where all invocations in a scope execute synchronously. This restriction implies that image sampling must occur in every possible code branch, prohibiting sampling in conditional statements with variable iterations unknown at compile time. Therefore, we adopted a fixed number of samples along the ray for the SSR implementation.

Below is an image showing the stages of the rendering pipeline, depicting the flow from shadow mapping to the final composition that's presented on the swapchain image.



Conclusion

In conclusion, our project successfully utilized WebGPU technologies to create a multi-stepped rendering pipeline for "Bomb Arena," including advanced features like a shadowmap, particle system, and Screen Space Reflections (SSR). Through this process, we gained knowledge about the complexities of video game development, especially in optimizing fun and playability. While our game implements a complex rendering pipeline and engaging gameplay for diverse audiences, future improvements in gameplay elements, models, textures, and adding animations would further enhance the gaming experience.

Sources

WebGPU Samples: https://webgpu.github.io/webgpu-samples/

Screen Space Reflection: https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html

MSDF Atlas Generation: https://github.com/Chlumsky/msdf-atlas-gen

UE4 MSDF Font Generation: https://sumfx.net/ue4-msdf-font-generator

Models and textures

Blender: https://www.blender.org

Sketchfab: https://sketchfab.com

TurboSquid: https://www.turbosquid.com

Itch.io Game Assets: https://itch.io/game-assets

Sound effects and music

FreeSound: https://freesound.org

Pixabay: https://pixabay.com

Appendix

The full source code is available at the following GitHub repository:

https://github.com/VoidRune/BombArena