DirectX PCG Physics Dungeon

Introduction

Variety is paramount in exploration focused games. Many games use procedural content generation (PCG) to create dynamic variety. This application provides a gameplay experience that is generated through cellular automata PCG; collectibles for the user to find in a generated level; a physics system to interact with an object in the scene; sepia post-processing to change the visual flair of the level; and ImGUI support to make these features interactable. The GUI in-game allows the player to tweak parameters of the PCG as well as the physics system.

Description

Overview/Controls

This application was created using C++ in Visual Studio 2022. DirectX11(Microsoft) and ImGUI (Ocornut, 2020) were external libraries used in development. The goal of the game is to collect golden cubes laid out across a dungeon while observing and interacting with a physics object. Players interact with the world by adjusting parameters in the ImGUI interface in addition to using the following controls scheme:

- “W” and “S” to move forward and backward respectively.
- “A” and “D” to rotate the camera left and right.
- “Spacebar” to kick the object.

There are three primary features prominent in this application: PCG, post-processing, and a physics system.

PCG

Procedural content is generated on request by pressing the “Generate” button available in the GUI. The content being generated is the layout of the dungeon and the placement of the collectibles across the new location.

This application uses a generative method called “cellular automata” to iteratively improve the layout of the dungeon. A 2D grid representing the XZ-axis is created and seeded randomly based on ImGUI provided parameters. The seeding selects XZ coordinates randomly to be walls of the level. The probability of this selection is available for the user to change and experiment with via the GUI. Using the initial seeding, the algorithm iteratively loops over every grid location and checks a given coordinate’s Moore neighborhood. If the number of neighbors that are walls is higher than a certain threshold, the observed tile is set as a wall. Walls are detectable by observing the y coordinate at any location. By setting the y-coordinate to a large value, the terrain slopes between the floor and this high point thus creating the wall. The scale of the room is unchangeable; however, the threshold and number of iterations are interactable.

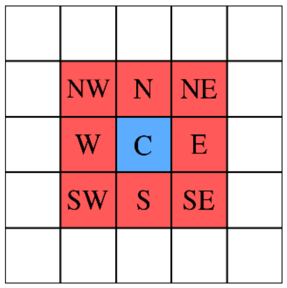

Figure 1: Moore neighborhood (Wikipedia 2022)

Post-processing

Post-processing is a method used to allow additional shader operations to occur on the scene before it is presented to the player. This application uses post-processing to apply a sepia filter to adjust the coloration of the scene. The implementation was created by referencing DirectXTK wiki for post-processing (Walburn, 2018).

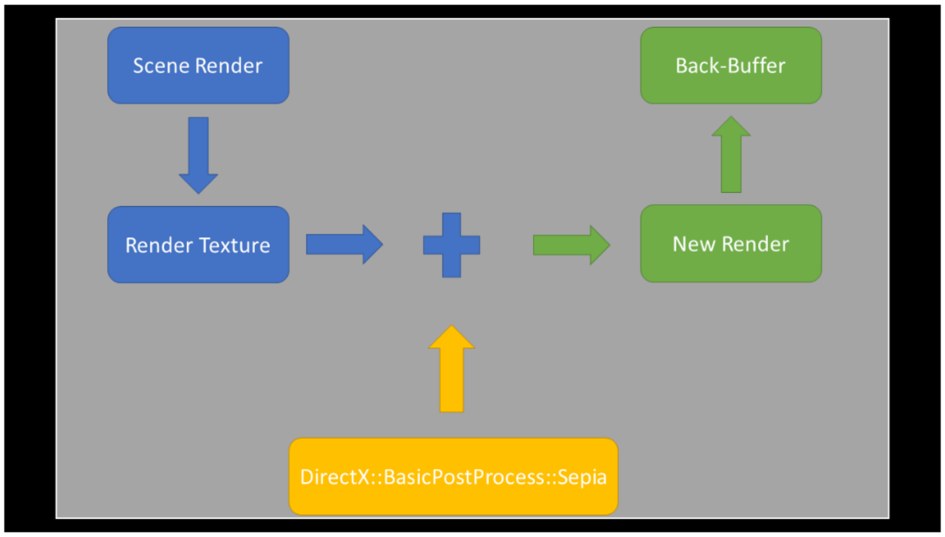

Figure 2: Post-processing functionality

Instead of rendering the scene directly to the back-buffer, this process first renders to a render texture. From there, a sepia filter is applied using DirectX effects functionality. With the new scene effect applied to the render, the result is drawn to the back-buffer and displayed in the application. This feature results in a greyscale coloration for the objects in the game.

Physics

The physics system is housed inside a separate class for this project.

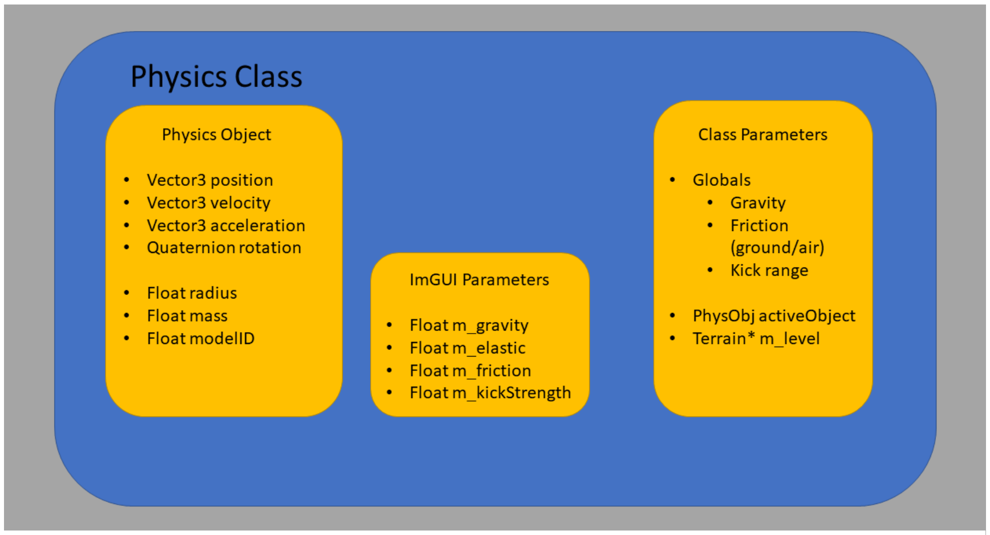

Figure 3 Physics Class architecture

The system has its update method called every tick and continuously calculates the forces and movement associated with the active physics object. The active object is a sphere that the player can spawn at will using a button in the GUI. The ball is spawned with a mass that the player can specify and a fixed diameter of one. The mass is used for physics calculations and the diameter is used for collision detection. To properly simulate a physical object in this environment, there were three main aspects this project prioritized: gravity, friction, and elasticity. All of these features are adjustable by the player so that physical behavior can be tweaked and observed. The equations listed below were used to generate the physics of the application.

$$Force = mass * acceleration \quad [1]$$

$$mass_1 * velocity_1 = mass_2 * velocity_2 \quad [2]$$

$$Force_{Friction} = FrictionCoeff * Force_{Normal} \quad [3]$$

Gravity is defined globally and scaled by the player provided parameter. This applies constantly to the object so a force [1] is applied on every update to always keep the object under the influence of gravity.

Collisions are traditionally handled by detecting if objects would be within the dimensions of another and setting them to an old position if they are. For the physics system, further elaboration was introduced. Collisions with the physics object are partially elastic, and as such observe the conservation of momentum. Since the mass of the object is never increasing or decreasing as a result of a collision, the only aspect to consider is the velocity [2]. If a collision is detected, the physics system determines the axis it occurred on and inverts the velocity on this axis. Scaling the new velocity by the coefficient of elasticity forces the object to lose some velocity in this exchange. The elastic collision feature allows the object to bounce off the walls and floor as forces push it around the room. The coefficient of elasticity is available in the GUI for the user to experiment with.

Friction occurs in an opposite direction to the movement of an object as a force proportional to the normal force and the coefficient of friction [3]. In this application the player has control over the coefficient of friction via a GUI. There are two types of friction, air resistance, and surface friction. This project determines the correct scenario by checking whether the object is currently colliding with any surface. If not, air resistance is applied as a slight force opposite to the direction of movement. Finding this direction is achieved by normalizing the velocity vector for the object. If the object is colliding surface friction is applied instead. The axis the collision occurs on does not undergo friction since it is applied perpendicular to the surface. The normal force is then calculated as the force of the object in the direction of the collision. This is then applied to the aggregate forces acting on the object. Friction also required further guarantees. Friction cannot change the direction of movement an object undergoes. With that in mind, friction is clamped to only apply if the direction of movement will not change as a result.

Visual feedback was implemented in the form of rotations on the sphere object. When a ball is pushed, the object undergoes a rotation as a visual cue of the action. For this rotation to stay consistent, incremental rotations need to be saved and compounded. To avoid gimbal lock, the rotations are calculated as quaternion rotations around the axis perpendicular to the movement. These incremental rotations are saved to the object and converted to matrices when multiplied into the aggregate.

Efficiency

The project is set up to scale with further implementation by being extremely modular with the functionalities. The physics engine takes three-dimensional vectors as parameters and contains a pointer to the terrain class to facilitate calculations happenings inside the module.

The nature of the grid-based terrain lent itself to an efficient form of collision detection. Instead of using AABB collision to check the boundaries of all objects in the scene, simply checking the neighbors was sufficient. The implementation thus saved a lot of overhead on collision detection.

Further efficiency improvements could be made to the abstractness of the class. The pointer to the Terrain object limits the functionality to this specific implementation. Abstracting the terrain to a simpler data structure would allow the modularity to improve. Additionally, the collisions that require the Terrain class could be moved into the physics engine itself, or, preferably, a separate collision class.

Nevertheless, the organization could be improved. A vast amount of functionality was implemented within a tight time constraint. Features such as physics collisions have duplicate checks occurring to acquire the respective axis. This could be improved to reduce the overhead on each update tick.

The cellular automata is also a rather inefficient implementation. While a large quantity of iterations ultimately reduces the complexity of the level, future use may desire a vast number of iterations. Iterating through the entire terrain every loop is inefficient and creates a large overhead. Isolating the seeded grid spaces and only examining the neighbors could be implemented using a depth first search technique. This type of improvement would greatly reduce the overhead needed for PCG iterations.

Reflection

Cellular automata is a useful technique for procedurally generating labyrinthian caves and levels. However, there are limitations that occur from applying this process of PCG. Fixed structures such as passages and rooms are not created through this process. Any type of structure desired would need additional functionality to be applied post cellular automata. Additionally, there is a very particular sweet spot for generation. The threshold and iterations scale the complexity of the level exponentially. A threshold of five neighbors iterated over five times generates a vastly different layout to a threshold of four neighbors iterated five times.

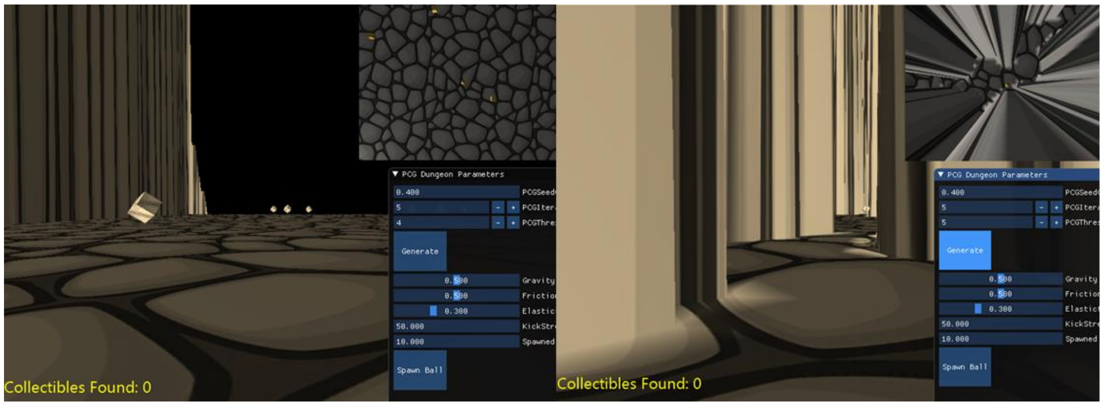
```
▼ PCG Dungeon Parameters
0.400          PCGSeed
5      - +     PCGIter
4      - +     PCGThre
Generate
   0.500       Gravity
   0.500       Frictio
   0.300       Elastic
50.000         KickStr
10.000         Spawned
Spawn Ball
```
```
▼ PCG Dungeon Parameters
0.400          PCGSeed
5      - +     PCGIter
5      - +     PCGThre
Generate
   0.500       Gravity
   0.500       Friction
   0.300       Elastic
50.000         KickStr
10.000         Spawned
Spawn Ball
```

Figure 4: Cellular Automata with different thresholds 4(left) and 5(right)

A physics system is easy to implement but difficult to keep track of. This means that the chance of compounding errors greatly increases. This application initially scaled the accelerations and forces by the time change between ticks, delta time. However, this caused cascading problems as the frequent multiplications meant the result was instead being scaled by an exponential version of delta time.

The idea of physics objects carrying over attributes such as velocity and position are instrumental in any physics system. However, with the force-based implementation found in this application, carrying over acceleration is not the proper method. Initial testing with aggregate acceleration created a scenario where gravity and friction would be applied multiple times per tick. Since acceleration is the result of the sum of forces divided by the mass of the object [1], the solution is to instead zero the acceleration at the start of every update and apply it instead reapply the forces every tick. In a scenario without friction, this could be simplified to initializing the acceleration as gravity every update. However, the more complex aggregate that friction creates necessitates a new acceleration to be computed every update.

Conclusion

The physics engine resulted in a complex and interactive addition to the procedurally generated level. The complexity of the calculations required a better understanding of debugging methods to find errors as they arose. Additionally, prior practice with quaternions was instrumental to implementing aggregate rotations on a sphere. Regrettably, the application was not able to feature multiple simultaneous physics objects due to time constraints. Nevertheless, the current application is very good at what it does. The physics system combines well with the PCG terrain. Players get to experience a sandbox style display of procedural content while influencing the generation directly by using the ImGUI display. The results of the generation can then be observed on the physics system. The physics object can be bounced off newly generated pieces of terrain and pushed around the room by the player. In conclusion, this application showcases a robust physics system supported by a dynamic level created using cellular automata.

References

Bett, M. (2021) *CMP505 5 - Procedural Dungeons*. Available at: https://web.microsoftstream.com/video/73623de7-c7d3-4cd1-8f18-bb56c2d12d31 (Accessed: 23 May 2023).

Free vector: Cartoon stone texture (2016) *Freepik*. Available at: https://www.freepik.com/free-vector/cartoon-stone-texture_976364.htm#query=cobblestone%20texture&position=4&from_view=keyword&track=ais (Accessed: 23 May 2023).

Harp, K. (2018) *Photo by Katie Harp on unsplash*, *Yellow and white area rug photo – Free Texture Image on Unsplash*. Available at: https://unsplash.com/photos/Em96eDRJPD8?utm_source=unsplash&utm_medium=referral&utm_content=creditShareLink (Accessed: 23 May 2023).

Microsoft *Microsoft/DirectXTK: The DirectX Tool Kit (aka DirectXTK) is a collection of helper classes for writing directx 11.x code in C++*, *GitHub*. Available at: https://github.com/microsoft/DirectXTK (Accessed: 23 May 2023).

Moore neighborhood (2022) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Moore_neighborhood#/media/File:Moore_neighborhood_with_cardinal_directions.svg (Accessed: 23 May 2023).

Ocornut (2020) *OCORNUT/IMGUI: DEAR IMGUI: Bloat-free graphical user interface for C++ with minimal dependencies*, *GitHub*. Available at: https://github.com/ocornut/imgui (Accessed: 23 May 2023).

Walbourn, C. (2018) PostProcess. Available at: https://github.com/microsoft/DirectXTK/wiki/PostProcess (Accessed: 23 May 2023).

Special Thanks to Matthew Bett for providing the framework for this application.

Video Demonstration available at:

https://youtu.be/95VvZEThu14