UNIVERSITY of ABERTAY DUNDEE

School of Design & Informatics

Session 2019/20

CMP505: Advanced Procedural Methods

Labyrinth Generation using Cellular Automaton

Daniel Zammit Student ID 1905316

MSc Computer Games Technology

Table of Contents

Introduction

The work conducted in this project revolved around the creation of a 3D DirectX11 scene using the DirectX Toolkit provided by Microsoft (Walbourn, 2019). The aim was to implement and utilise an advanced procedural technique that can be used as a gameplay element in a released game. In order to achieve this goal, cellular automation was implemented to create a traversable labyrinth were the objective of the game is to find and collect the treasure. A scene was created were several boxes were placed in a grid formation. Most of these boxes were converted to obstacles. The player was able to move around the labyrinth. However, when the player collided with any obstacle, they are returned to the spawn point. When the treasure was collected, a new level was generated. A skybox was also added for immersion.

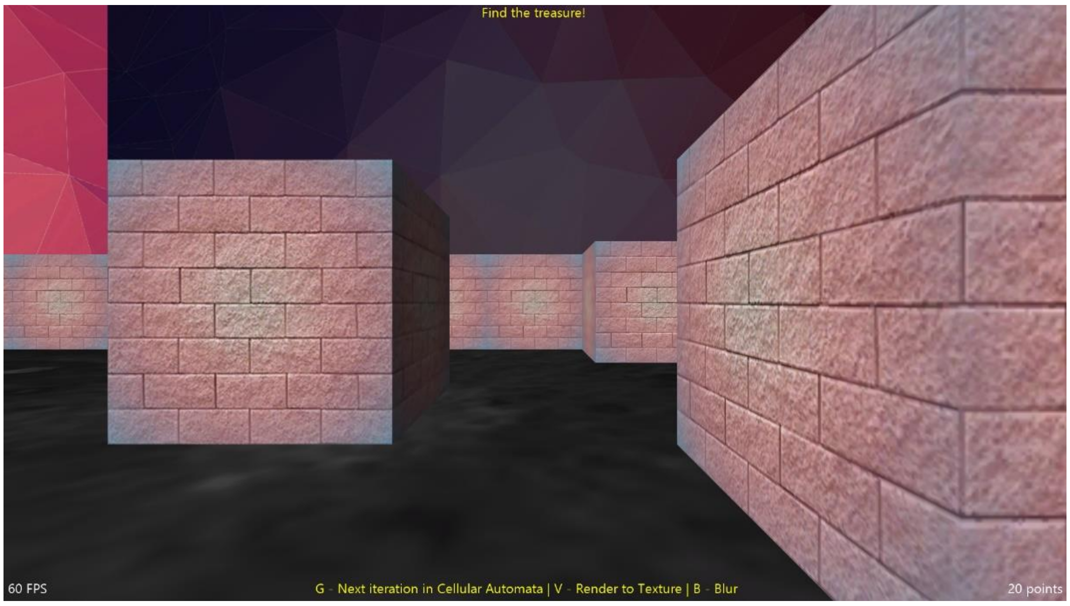
Find the treasure!
60 FPS
G - Next iteration in Cellular Automata | V - Render to Texture | B - Blur
20 points

Figure 1 - Labyrinth Scene

The scene objects were initialised using DirectX11 API calls using C++ programming language. A point light was also added using HLSL based on Blinn-Phong reflection model to brighten the scene. The player camera has free movement in two-dimensional space, as well as 360° rotation. While playing the game, it is possible to view the next iteration of the cellular automaton in order to find the treasure more easily. Blur was implemented as a post-processing effect, which was applied to the main render target.

Feature Implementation

User Input

The game controls were bound to the keyboard with movement being set to the standard WASD scheme. The G, V and B keys were used for next cellular automaton iteration, mini-window display and blur toggle respectively.

In order to achieve this, an Input class was created to handle all the user's input and pass the information to the Game class (main class of the application). Using this approach, it was easier to add extra input functionality during development. The input was then used to toggle checks and enable the desired functionality in the Update method. With this information, the camera view matrix was manipulated to present the new view each frame.

Cellular Automaton

The main feature of the developed game was procedural generation using cellular automaton (CA). The idea behind CA is defined as a model of a system with cell objects having characteristics identical to Conway's Game of Life. A cell lives on a grid, were each cell has a neighbourhood and a state, alive or dead (Shiffman, 2012). Depending on the neighbourhood states, a new state is assigned to that cell. The rules are applied to the next generation of cells.

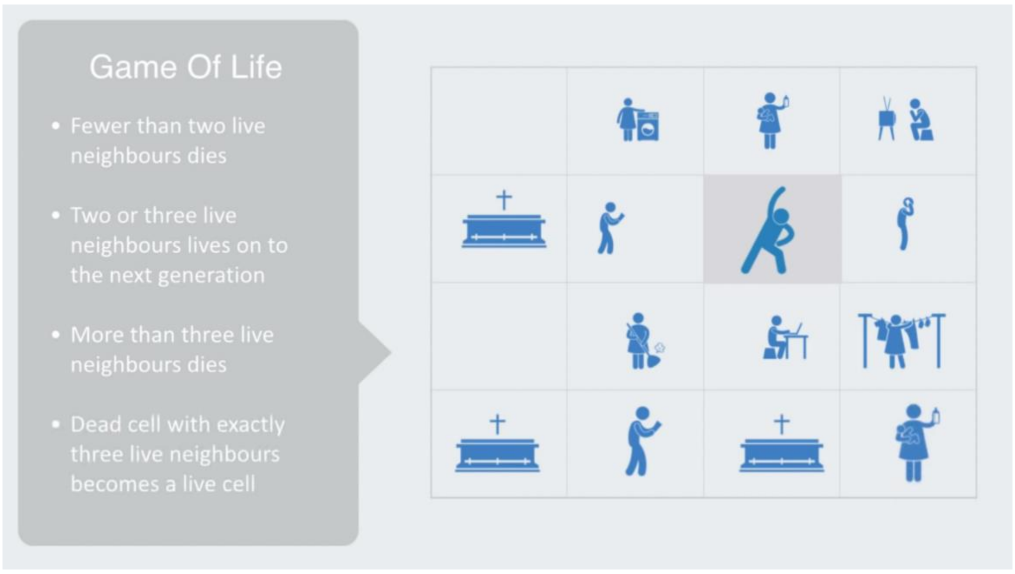
Game Of Life
• Fewer than two live neighbours dies
• Two or three live neighbours lives on to the next generation
• More than three live neighbours dies
• Dead cell with exactly three live neighbours becomes a live cell

Figure 2 - Conway's Rules for Game of Life

Grid creation was achieved by calling an initialisation function in the Grid class. The initialisation function was responsible for populating a 2D cell matrix with randomly assigned states. To take advantage of object-oriented programming, a Cell class was created to describe a cell object and store its information. To randomly distribute and seed the cell matrix with cells of different states, the built-in rand function was used. When creating a grid, each cell was given either an alive or dead status.

In order to integrate gameplay with cellular automaton, the algorithm was modified to allow player and treasure cells to be placed on the grid. Using an array of integers instead of booleans provided more flexibility with cell state assignment. To ensure that the resultant grid was suitable for the purposes of the game, checks were included to ensure that both the player and the treasure cell have been placed in the labyrinth once. The next generation of cells was calculated based on the neighbourhood of the current generation. As the grid is based on a 2D matrix, Moore neighbourhood was used to retrieve the state of neighbouring cells. With probability being used to seed the grid, it was necessary to implement a method to ensure that the grid was solvable and that the player can reach the treasure. This was achieved using A* pathfinding.

A* Search Algorithm

As the labyrinth was built using a 2D matrix which forms a grid, using A* pathfinding was a natural choice. The unique aspect of this algorithm is the efficiency in finding the optimal path based on the cost of each node or cell in this case ('A* Search Algorithm', 2016) . On each step to find the target, the algorithm selects the node with the lowest value (f) of the sum of the movement cost to move from the starting point to a given cell on the grid (g) and the estimated movement cost to move from that given cell to the final destination on the grid (h). In order to achieve this, the Cell class was modified to contain these new variables. A new class named AStar was created. An AStar instance is initialised in Grid class and it is used to check for solvability.

The implemented A* pathfinding in this project was based on A* Search Algorithm (2016), with some minor modifications to integrate the existing codebase. In this approach, two lists are created using the set data structure and boolean hash table for open and closed respectively. Once the lists are initialised, the starting cell is placed on the open list and while the open list is not empty:

```
1. Find the cell with the least f on the open list and call it q
2. Pop q off the open list
3. Generate q eight successors and set their parents to q
4. For each successor
      a. If successor is target cell, stop search
      b. Calculate successor g and h based on the sum of q.g value and distance
         between successor and q, for g, and calculate h using distance from target cell
         to successor. successor f is calculated by the sum of g and h
      c. If cell with same position as successor is in open list which has a lower f value
         than successor, skip
      d. If cell with the same position as successor is in the closed list, having a lower f
         than successor, skip this successor, else add the cell to the open list
   endloop
5. Push q on the closed list
6. End while loop
```

The distance formula is calculated using Euclidean distance for approximation of h.

$$h = \text{sqrt} ((\text{current\_cell.x} - \text{target.x})*2 + (\text{current\_cell.z} - \text{target.z})*2)$$

As mentioned above, the resultant outcome from the A* algorithm is used to check the solvability in the grid to ensure the labyrinth is solvable.

Post-Processing

The player has an option to toggle blur while playing. The effect was implement using post-processing provided by DirectXTK library (Walbourn, 2018). In order to render the blur effect, a render to texture target was initialised. A new method named Blur was created to render the objects in the scene to a new RenderTexture target and to reset the render target to the original back buffer. The post processing is applied after resetting the render target. This allows more than one post-processing effects to be layered together before rendering the final image.

Evaluation

The application managed to fulfil the objectives set out by the assessment brief. During development, a great amount of time was spent thinking about class management and hierarchy. Using C++ as a programming language facilitated both management and optimisation. The code is well structured and well documented, highlighting the thought process throughout the development of the game. In future work, an enhanced strategy for collision checking can be introduced to check the current position of the player with the nearest cell on the grid, and test against all active cells in that grid. Currently, collision checking is takes into consideration the global coordinates and they are given to AABB testing method.

With regards to the procedural generation content, implementing cellular automaton with DirectXTK was a great challenge. However, the greatest challenge was ensuring that the generated labyrinth was solvable. Although there were some issues with the implementation of A* algorithm with the rest of the framework, the result is quite satisfactory. The ability to manipulate the labyrinth by calling the next generation of cellular automaton create a fun dynamic for gameplay.

Limitations

The game has a minor issue where the solvability of the grid is calculated on the old position of the player in the grid. Therefore, it might be the case where in rare cases, the grid that is generated is not solvable when the player cell has been changed. This was a design limitation during the development of the game. Future work can focus on checking whether the grid is solvable if the player has been moved due to grid regeneration in the case it is not solvable. There is also a minor bug with player score which is most probably related to the same issue.

Reflection

During the development of this project, several lessons were learned, with the most important being time management. The project can be split into three segments: cellular automaton, A* pathfinding and research. Out of all three, most of the time was spent on researching procedural generation in games. Cellular Automaton was chosen for its simplistic understanding and various applications. Particularly, game development applications. They can be used in tower defence games to calculate path from player and enemy bases and city building simulators. The development process was a great test of skills and an opportunity to gain a better understanding on the C++ language.

References

'A* Search Algorithm', (2016) *GeeksforGeeks,* -06-16T23:49:37+00:00. Available at: https://www.geeksforgeeks.org/a-search-algorithm/ (Accessed: May 12, 2020).

Shiffman, D. (2012) *The nature of code : [simulating natural systems with processing].* S.l.]: Daniel Shiffman.

Walbourn, C. (2018) *PostProcess.* Available at: https://github.com/microsoft/DirectXTK/wiki/PostProcess (Accessed: May 12, 2020).