ECE 4122/6122 Final Project Report

PuzzleGL, the OpenGL Jigsaw Puzzle Game

Fo-Yao Alessou <falessou3@gatech.edu>

Palak Choudhary <choudhary.palak@gatech.edu>

Akshay Nagendra <akshaynag@gatech.edu>

Joseph Stevenson <jstevnson33@gatech.edu>

Paul Yates <paul.maxyat@gatech.edu>

Submitted May 3rd, 2018

Introduction and Execution Instructions

PuzzleGL is a jigsaw puzzle game using C++ and OpenGL libraries. It allows a user to play a jigsaw video game on their computer by having the user click and drag pieces in a graphical window to re-assemble a scrambled image. The target image is displayed for 5 seconds before it is scrambled, and then the user must figure out how to get back the original image by assembling the pieces in the correct order with a time limit of 3 minutes.[1] If they do so correctly, the game will advance onto the next stage, where more pieces are introduced to make it harder. PuzzleGL currently supports three increasing puzzle levels and once a user completes all three stages, they are greeted with a victory screen and the game exists. If a user fails to assemble the correct image within 3 minutes at any stage, the user is presented a "Game Over" screen and the game exits with the highest level achieved displayed in the console.

In order to run this project, an environment that supports CMake Version 3.9, C++ 11, and OpenGL 3.3 is required. If using the Jetbrains CLion IDE, running the code is as simple as opening the project from the project dir (*PuzzleGL/*) and clicking Build then Run or Run directly[2]. Since the project uses GLAD, the user must verify that the version of GLAD works with their computer. The game has been tested on Windows operating systems and functions as expected.

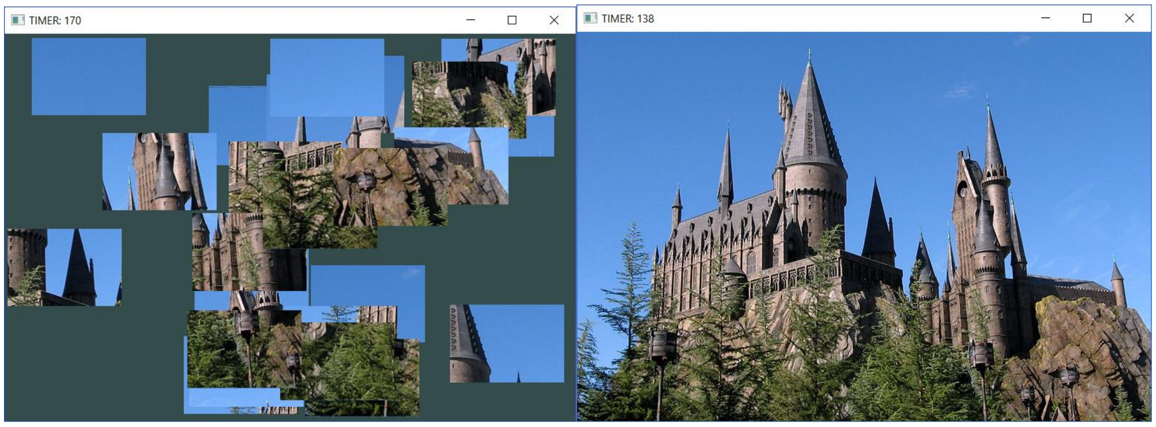

Figure 1. Example screenshot of PuzzleGL showing the scrambled pieces (left) of an image needed to be constructed by the user. The solution can be seen on the right. Note the time left can be seen in the window title.

Implementation Details

The core implementation of PuzzleGL uses OpenGL, GLAD, and GLM. Each puzzle piece is represented as a pair of triangles forming a rectangle, which use the same 6 vertices across all puzzle pieces, combined with a transformation unique to each puzzle piece. The puzzle pieces form an N by M grid which fills the screen, where N and M increase as each successive puzzle is completed, but starts at 4x4.

[1] *PuzzleGL* also supports a DEBUG_MODE which is for instructor evaluation of the program. In the *main.cpp* file, there is a parameter highlighted by comments labeled *DEBUG_MODE*. If set to 1, then the image will remain unscrambled

[2] Warning: CLion tends to suppress stdout when there are many messages being displayed so running the project using the *debug* button might be more helpful since all the messages can be observed in the CLion console

The puzzle image is also defined per level, and the window dimensions will be sized to match the dimensions of the chosen image.

The texture is loaded in a few steps. First, the "stb_image_h" library is used to load an image into a character array. This array is then passed into a loadTexture function that does everything necessary to apply that image to a texture. Included in this are creating the texture, setting the texture parameters, loading the image to the texture, and generating the mipmap. Additionally, it frees the image array from memory. Next, the vertex and texture shaders has to be modified to work with work with a texture. The vertex shader applies a texture offset. This offset allows each puzzle piece to have a different part of the image. This offset in calculated before displaying the texture and applied to each piece so every piece has a different offset. This process is repeated for each level of the game.

In order for the user to complete the puzzle game, mouse click and drag functionality was implemented. When the user clicks within the game window, the game checks to see if any puzzle pieces lie below the mouse cursor. If more than one puzzle piece is below the cursor, the piece with the greatest z value is chosen as the clicked piece. When a piece is dragged, its location is not simply set to the location of the mouse cursor, as this would cause the piece to be dragged only from the center of the piece. Instead, an offset is calculated as the difference between the x and y positions of the mouse and the puzzle piece at the time at which the puzzle piece was clicked. This allows a puzzle piece to be click-dragged in an intuitive manner, even if only a corner of the piece is visible to the user. Finally, dragging pieces offscreen is prevented by bounding the piece x and y locations to between -1 and 1.

When a puzzle piece is clicked, it is immediately displayed as "above" all other pieces, i.e. if a piece is partially obscured by other pieces, it will become completely visible to the user when clicked.This is achieved by sorting the pieces by z value. Instead of using a z value between -1 and 1 to draw the pieces and have OpenGL handle masking, the pieces were organized into a sorted vector. The pieces are drawn to the screen in ascending z order, such that the piece with the highest z value is drawn last, and is thus shown as being "above" all other pieces. This has the benefit of making the piece clicking feature very simple: the z value of a clicked piece is simply set to greater than the z value of the last piece in the vector, then the vector is sorted. In this way, z values may be completely arbitrary integers, serving only to designate an ordering between pieces. This sorting also has the added benefit of making click detection more efficient: the vector can be reverse iterated across, and the first piece which lies under the cursor is chosen as the clicked piece, as this piece has the highest z value of all pieces which lie beneath the cursor.

A jigsaw puzzle uses pieces which stay together once they have been placed together. Thus, this game requires that pieces can be snapped together. Snapped pieces may then be dragged around the screen and snap to the top screen layer as if they were a single piece. In order to achieve this, each piece has a vector of group members. A group member is defined by a puzzle piece and an (x,y) offset to the puzzle piece which owns the group. When pieces are grouped, any click and drag operation which applies to the clicked piece is also applied to every member of that piece's group, while maintaining the original spacing between pieces via the offset value. Note that for correct functionality, when piece X has a group member Y, then piece Y's group should also contain X.

Through the course of the game, pieces are grouped when they are placed within proximity of one another. Whenever a piece or group of pieces is placed at a certain location (i.e. after mouse button released), each piece in the group will check if it is next to one of its neighbors. Each piece has four neighbors which should be an equal offset away in the four cardinal directions. Once a piece is determined to be close enough to its correct neighbor, its (x,y) location coordinate is updated such that it snaps to the neighbor with no whitespace. The trickier case is when two distinctly grouped pieces are pushed together with the goal of creating one massive grouped piece. Figure 2 highlights this case.

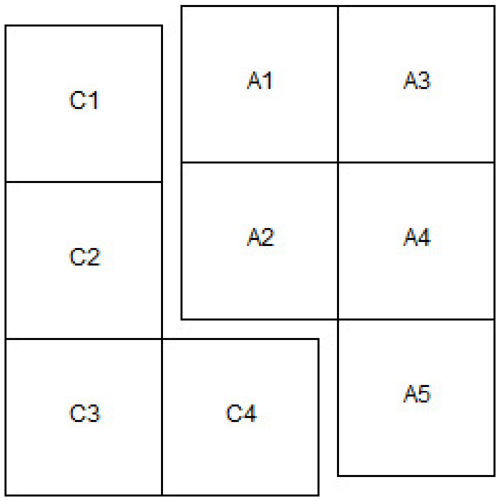

Figure 2. Situation where two grouped pieces (**C** and **A**) are trying to be joined together. **C** and **A** share multiple edges for contact causing potential complications if all pieces of **C** and **A** are not updated correctly.

As per Figure 2, let **C** be the group of pieces that form the candidate group of pieces that the active group of pieces (**A**) is trying to connect to. The algorithm basically iterates through all the pieces of **A** and whichever piece is determined to be closest to its correct neighbor in the **C** group first shifts its position such that the particular **C** piece and the **A** piece have no whitespace. In addition to shifting its own position, the **A** piece determines the offset by which it was shifted in order to shift to its **C** neighbor piece and forces every piece in **A** to shift by the same amount. This prevents a jitter effect where some pieces of **A** connect to **C** but it is not reflected in the offset value, as discussed earlier. Finally, whenever pieces are grouped together, they all add one another to their respective group list like a strongly connected graph (cliques).

To start a jigsaw puzzle, all pieces are separated and mixed about. Thus, this game has a scramble feature. The user may press the “S” key at any time to randomize the x and y locations of each piece, and to ungroup all pieces.

There are three puzzles that the player must finish in order to complete the game. The level of difficulty increases with each level, with larger number of puzzle pieces. As soon as the user finishes a puzzle, the next puzzle is displayed. On completion of all three levels, a “YOU WIN” message is displayed for ten seconds after which the program exits. However, as can be seen in Figure 1, a countdown timer is implemented that counts down from 180 seconds and constantly updates the GLFW window so the user can see the how much time they have left to solve the puzzle. If time expires, a “GAME OVER” message is displayed for 7 seconds and the highest level achieved is printed to the console so the user can see how far they got.

Team Role Details

The design of PuzzleGL was split into multiple different modules such that each person in our project team could work on the design of the game concurrently as much as possible. In order to help with merging code and versioning, GIT was utilized and allowed solving bugs that appeared in versions of the code to be efficiently rectified.

Paul Yates

This group member created the puzzle piece grid with a dynamic number of pieces. He designed the click-drag and piece grouping features, and implemented screen-size to image dimension matching. He also assisted in the creation of the scrambling and piece snapping features.

Akshay Nagendra

This group member implemented the snapping code and algorithm for grouping pieces together. He also generated the initial project infrastructure to verify whether or not the OpenGL/GLAD setup would work for all team members. He also implemented the countdown timer that is constantly updated in the GLFW window to show how much time the user has left to solve the current stage, as well as the “Game Over” logic to terminate the game, showing the “GAME OVER” message, and print the highest stage achieved by the user.

Palak Choudhary

This group member designed the multiple stages feature of the game. She implemented the feature to increase difficulty level of puzzles at each stage. She also designed the scrambling feature, which randomizes the positions of all puzzle pieces, effectively creating a starting point for the player.

Joseph Stevenson

This group member implemented the image and texture for the puzzle pieces. This applied the picture to each of the piece in the correct display order. He also helped to write both the report and proposal.

Fo-Yao Alessou

This group member partially implemented a count-up timer for the game. The timer display the total time its took for the player to complete each level of the game.

Conclusion and Discussion

The PuzzleGL project provided the team a great opportunity to incorporate OpenGL with gaming. It also provided a new set of debugging challenges, such as using matrix transformation for movement of pieces, keeping track of a piece's location and state. One challenge that arose during the snapping code implementation was the issue of jitter, as briefly discussed in "Implementation Details." Originally, not all pieces of the **A** group (See Figure 2) were moved, and this caused the actual location of some of the **A** pieces to not match what would be expected by measuring its offset to a reference piece. A simple solution was introduced at the beginning by forcing pieces to be quite close to be grouped together (i.e. a smaller error/whitespace threshold), but this has the adverse effect of infuriating the users at the time since human precision is not always at the pixel level. After highlighting specific cases where the jitter was visible, the bug was fixed and all pieces of the **A** group were forced to move upon a snap, causing no jitter to occur any longer.

If more time was allocated for the project, future work would entail experimenting with different jigsaw piece sizes (e.g. having the classic jigsaw puzzle piece shape), rotation of pieces, adding new game features to enhance the experience (e.g. slowing down movement, distortion effects, etc).