Gorilla, an Interactive Go Application
by Drew Wright and Benjamin Streb

1: Introduction:

Go is an ancient board game originating a few thousand years ago in China. It is a game in which both players take turns placing stones on a board trying to surround space, and capture each others pieces. It is a game known for its elegant rules and appearance, and it is one of our favorite games. However it can be a little tedious to actually place and pick up pieces from the board without moving other pieces on the board. Of course, many others before us have created applications to play go, but the majority of ones we've seen are simple 2D GUIs. Our goal with this project was to create an interactive application that would try to imitate the Go experience while taking care of the less fun elements that could be automated, like picking up chains of captured pieces. We also wanted to create some visualizations that could be useful for newer players to help them in understanding the incredibly abstract game that is Go.

We decided one of the key factors in the experience of Go is the large amount of time that passes while you wage a war of wits with an enemy, something that we decided we could show by having a sun in the sky move slowly as the application ran. In order to achieve this effect we realized we would need to use shadows, as the shadows cast from the pieces to the board, as well as the board to the ground would change with the rotating “sun”. Without shadows this idea of the passing of time would not be as apparent.

However, we noticed that rendering shadows with a moving light source for a Go board could quickly run into issues. Unlike games such as Chess, as Go progresses the board gets full of more and more pieces, not less, and the sheer number of pieces that eventually are placed would prevent us from using a technique like ray-casting at rendering speeds quick enough for a real-time interactive application. We needed a technique that could efficiently, and relatively accurately, render shadows for a simple scene with dynamically increasing complexity. The technique that we came across, and decided to try to implement was Shadow Maps. This technique fit the bill for a method by which we could quickly render shadows for a complex scene. It is a technique so powerful that modern graphics hardware has incorporated several calculations used by Shadow maps right into the GPUs, and so to use this technique we had to delve into the land of OpenGL shader and buffer programming, a terrifying wilderness which we were not prepared for.

With a method for replicating the real Go experience laid out we next looked into a way we could help visualize the game state to create a useful tool for novice players. We decided that one of the most critical concepts in Go that a visualization could help with is control of areas of the board. Since pieces over control over areas much wider than their immediately neighboring spaces new players often place pieces in areas in which an enemy has too much control, or they bunch pieces too close together and, as a result, fail to get strong control of enough of the board themselves. We devised a system in which pieces emit a propagating control “energy” to nearby pieces which is counteracted by enemy pieces and enemy control “energy”. This energy could provide a quick and dirty estimation of board control while giving us something we could visualize.
Last but not least we needed to actual let players play go, so we want to code basic rules enforcement as well as letting players place pieces by clicking on the board.

2: Previous Works:

In recent years with the increasing power of graphics hardware and GPU programmability there has been a lot of research into various Shadow Map techniques. The paper that started it all was “Casting Curved Shadows on Curved Surfaces” by Lance Williams [Wil78] . The secret behind Shadow maps is that the algorithm first renders a scene from the viewpoint of the light source. It uses the depth of the first collision with the environment from that light source to create a texture. Then the scene is rendered from the Camera, and point are tested against this depth value to

determine whether to draw them in light or shadow.
Originally we wanted to implement Smoothies as described in “Rendering fake soft shadows with smoothies” by Eric Chan and Fredo Durand. [CD03] This was a technique that used information about edges to guess what the proper soft shadow would be, and would include this in the stored texture, but since we encountered issues with normal Shadow maps, we did not get to this.

3: Board Control Visualization:
The implementation of board control has two main portions. The first is the underlying data structure which represents the control “energy” on the board and the calculation of said energy. The second is the method by which we displayed this data to the players.

The control energy of a space is represented by a single float, with -1 being completely in the white player's control, 0 being under neither player's control, 1 being completely under the black player's control, and everything in between representing varying strength of control. The control of each space on the board is stored in an NXN array of floats. The control is calculated as follows: Whenever a new piece is placed on the board, it adds P(where P is 1 for black, or -1 for a white piece.) to the value of that space in the array. If the absolute value of that value is less then P it is raised to P. We then make two temporary data structures to store spaces that have undistributed energy which are stored in a queue as a pair of location and energy, and to store the spaces that have already distributed energy for this piece calculation which are recorded in a map. We then push this piece into our data structures. Next we loop until the queue is empty. During the loop we pop off the front of the queue storing it's energy which we halve and its location. We then run a calculation for each of the 4 adjacent spaces. If the adjacent space has yet to be visited we add our energy to it on the array. If that space was controlled by the opponent we reduce our energy to distribute by the amount of enemy control in that space. If there is any energy left we push that adjacent space, along with the undistributed energy into the queue. We also insert the adjacent space into the visited map. After this loop is finished, the board control is done updating for that piece.
Whenever a piece is captured we have to completely recalculate board control. This is done by resetting the board, and re-adding each remaining piece back to the board in order they were placed.

To display this data to the player we render colored squares overlaid onto the grid with colors that scale based on the float in the underlying array. We scale negative numbers to red and positive numbers to blue, with black belonging to neither player.

4: Game Implementation:
There were a few interesting problems we encountered in the game implementation. The first was getting a system to map the mouse position to a grid position. The next was the actual implementation of Go rules like piece capture, suicidal move prevention, and enforcement of the Ko rule.

To make the game playable with the mouse we first mapped the mouse position on the screen to world space. We then cast a ray from the camera position through the mouse position to the scene. The point that hit the plane level with the top of the board was mapped to the coordinate system of the board, and this value was passed to the game code for adding a piece.

In Go pieces are captured if there is no free space adjacent to it or any connected ally pieces. This is an interesting problem to do efficiently, and one which we gave up efficiency in exchange for simplicity. Our algorithm for piece capture works by calling a stillAlive() function on each enemy piece to the most recently placed piece. The stillAlive function is similar to the control map calculation algorithm, but instead of spreading to all adjacent spaces, it only pushes allied pieces into the waveFront queue(called chain in the stillAlive function). The moment the algorithm checks a space with no pieces it cuts out and returns that the original piece is still alive. If the waveFront

queue empties then no adjacent spaces were found in the chain and the piece is considered dead so the function returns false. We add each piece found to be dead to a vector. This vector is passed to a function that removes each piece in the vector from the board.

Since pieces die if there is no adjacent free spaces, it would normally be suicidal to place a your own piece into a piece completely surrounded by the enemy. In Go doing this type of suicidal move is usually illegal. The only time it is legal is if placing your piece there would deprive the surrounding enemy pieces of their last free adjacent space, thus capturing them. If this happens, you capture their pieces, and your newly placed piece survives. We implemented this rule by first checking if a newly placed piece captures any enemy pieces. If it does not, it then checks if any ally pieces are now dead. If it does find dead pieces then it reverts all changes made to the game since the beginning of the most recent placePiece call.

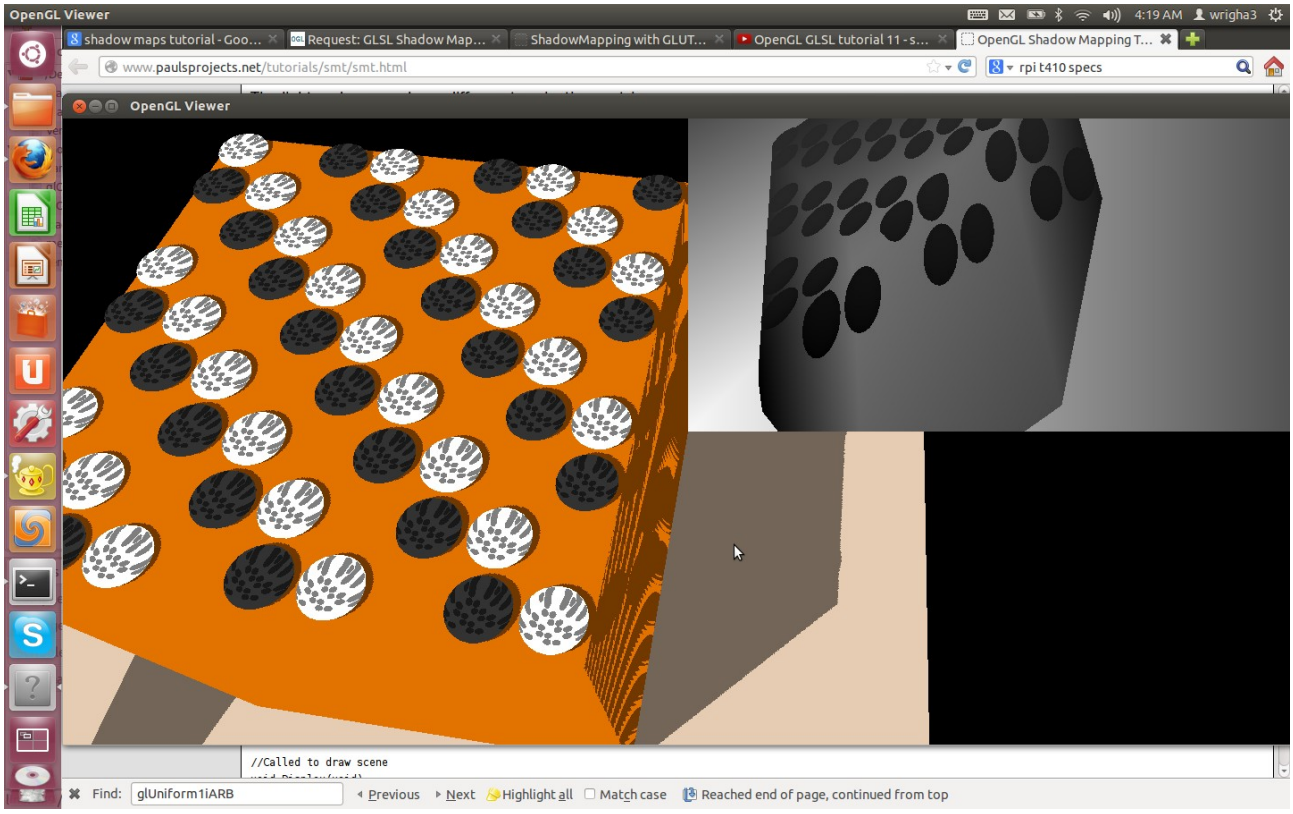
Lastly there is a situation that sometimes comes up in Go that is referred to as Ko. Ko occurs when you make a move that creates the same board position that your most recent turn ended on. In order to prevent games from going on forever a rule was created that prevents you from making a move that would recreate your previous board state. To enforce this rule we kept a record of the two previous board states, and updated them on each move. Since Ko can only occur with the capture of 1 piece I only check for Ko when a single piece is captured as the result of a move. After making this move we compare the resulting board against the board from 2 moves ago. If they are the same board then we rollback all changes made in this most recent placePiece call.

5: Shadow Map Implementation:
The implementation attempt was comprised of code added to our main rendering loop, a vertex shader, and a fragment shader. In our renderer we first added some code to setup the shadow texture that runs behind the scene to create our shadow map. Then we start the main loop. In the main loop we first reset the buffer, and enable face culling which is included to help prevent unnecessary self shadow. We then enable the Z-buffer. Next we perform a transformation to the viewpoint of the light, and render the scene. At this stage we don't bother with color as we are just trying to set the Z-Buffer. It is at this point that we set the shadow texture. We first need to make a bias matrix which will map our -1 to 1 values to a 0 to 1 range which is used by the depth buffer. Then we multiply our Model view matrix by our projection view in order to calculate where a point is calculated in the light-space of our shadow texture. We then move the view back to the camera view, and render the scene again, culling the back faces which we can not see, and the result, in combination with the sharers, should be that the shadows in the scene are drawn.
The vertex shader simply takes is value from the shadow texture.
The fragment shader is what we have taking care of the shadow calculation. I takes in a z value, or its distance from the light, and draws the fragment lighter or darker based on this retrieved value.

6: Challenges:
I think the biggest challenge we encountered was our own lack of experience with OpenGL and graphics hardware programming along with general technical difficulty. Many tutorials we read through on the Internet used functions that we couldn't seem to get working, or used libraries that we couldn't get working on Linux, or building properly. We actually partly implemented shadow maps at least three separate times before the current faulty rendition.
We also encountered difficulty attempting to create models for the pieces and tables since we didn't have experience with 3D modeling or the creation of .obj files.

7: Results:
The results were rather disappointing. After throwing many man hours at the Shadow maps, neither of us were able to get desirable results. The closest we got was a rendering that did make shadows

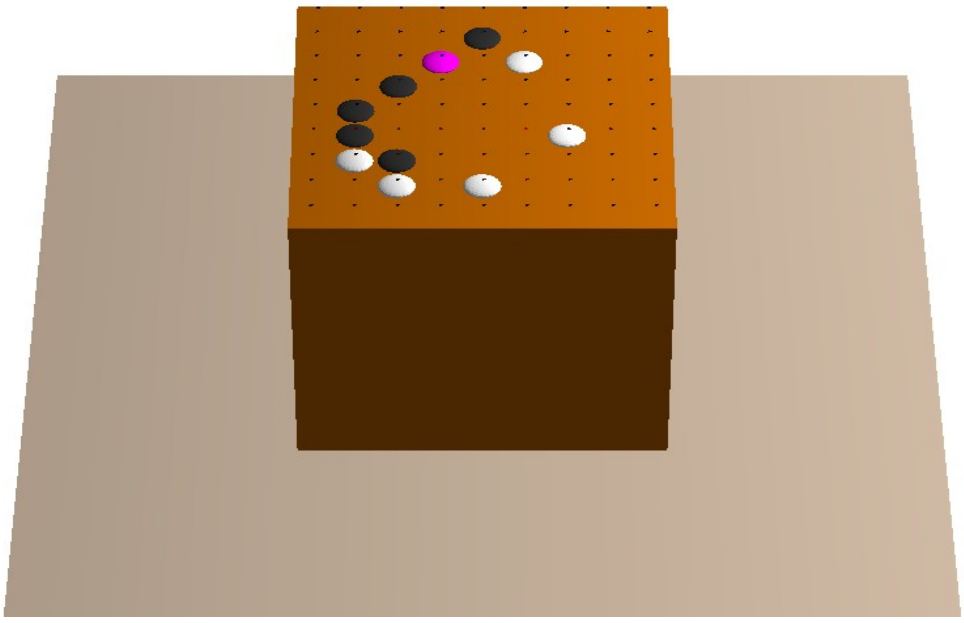
for the board onto the ground, and the pieces onto the board, but shadows of each of the pieces were appearing on each of the other pieces, and there were some major artifacts on the board shadow:



However, the faulty shadows did seem to behave relatively appropriately when we animated the light source, which leads us to believe that our current implementation is very close to an acceptable one. It was also cool to note that the shadow maps did actually render at perfectly interactive speeds so we achieved our goal of a shadow rendering method that could render in real-time.

The control map visualization was almost completed. It has a functioning back-end, and the colors of the rendered zones seems to be correct. At the end we encountered an issue with the mesh representing the control zone that we couldn't get completely resolved.

As for the gameplay, we never did get the mouse mapping quite right. As it stands, the pieces are placed at a strange offset. However, we do have pieces being placed on the board at the click of the mouse with alternating turns. We also make sure that pieces can only be placed on the board. We also have piece capture, suicidal move prevention, and Ko rule enforced.



All in all we spent an estimated 80 combined man hours on this project. Drew Wright spent around 20 hours on attempting to get a working implementation of shadow maps before moving on other areas in the project. He spent the remainder of his time implementing the board control algorithm, and visualizations, along with the underlying game-play code. Benjamin Streb spent approximately 12 hours getting a basic demo display put together including modeling the board and pieces. He spent the rest of his time trying to implement mouse interaction with the go board, and attempting to get the shadow map implementation working. He was also our merge master who managed to get everything compiled and running under extreme time pressure extremely successfully.

8: Future Work:
Obviously getting the basic Shadow map down would be awesome, as we can't really explore anything cool until that is taken care of. After that we could actually explore soft shadows, and maybe look into quick rendering of basic diffuse reflection off the pieces. The visualization of board control was an interesting problem and I think there is definitely room for improvement in our algorithm since having to recalculate the whole board after the removal of a single piece seems rather inefficient. Perhaps exploring some type of acceleration data structure which determines what pieces could possibly contribute to control in that area could be used to only recalculate smaller portions of the board. Doing this could make it scale well to arbitrary game sizes.

9: Bibliography:

[CD03] Eric Chan and Fŕdo Durand. Rendering fake soft shadows with smoothies.
In Proceedings of the Eurographics Symposium on Rendering, pages 208–218.
Eurographics Association, 2003.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces.
SIGGRAPH Comput. Graph., 12(3):270–274, August 1978.

Tutorials Referenced:

http://www.paulsprojects.net/tutorials/smt/smt.html

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

http://www.fabiensanglard.net/shadowmapping/index.php