CS 559 Project 2 Specification “Moshball”

Preface

While I have not yet graded Project 1, I anticipate the grade distribution will be bimodal. I expect a gulf between those that started early and worked continuously and those that put off too much until it was too late.

There are four graded components in this course: Three projects and one midterm. Each is normalized against the highest score achieved in the class in that component. Your individually lowest score is dropped. Your final grade is computed based upon the average of the remaining three.

If you did very poorly on project 1 and believe your subsequent grades will not improve, please consider dropping the course. Or if you did very poorly on project 1 and now realize the wisdom of working steadily on your future projects, then by all means continue.

Overview

The object of project 2 is to code the game of Moshball. This is a game I invented back when the notion of the mosh pit was relatively new. I combined the notion of mosh pit with the cliché “Be the Ball!” The result is Moshball.

You must hurl yourself (a ball) into a number of other balls (defined by command line parameter). Upon striking another ball, it starts a countdown timer which is influenced by the number of balls. The greater the number of target balls, the higher the initial value of the countdown timer.

The game is won when all of the target balls are in the “bumped” state. That is, all of the targets are still counting down their timers.

You must accurately and precisely calculate collisions and bounces. In the base version the target balls are fixed in their positions. It is a suggested bonus feature to cause the target balls, when struck to move in a way akin to billiards. If you implement this bonus suggestion, a moving target ball striking another target ball starts the second ball’s timer just as if you had struck the second ball directly.

Pedagogical Purpose

In implementing this project you must master:

- precise (2D line-circle and 2D circle-circle) collision detection in a fast moving game
- the use of textures loaded from image files (use DevIL)
- the use of textures for the purpose of rendering text (use freetype)

- the concepts of intricacies of the OpenGL programmable pipeline as represented by GLSL vertex, fragment and (bonus) geometry shaders
- the use of textures created using GLSL shaders
- the use of skyboxes
- the creation of multiple viewports showing different projects
- the use of off screen rendering (frame buffer objects) to create dynamic textures

Suggested bonus features:

- Billiards-like action as described above – adds bonus (but makes game easier). Note, last year’s class actually did billiards and you can hijack code from them (but you are responsible for their bugs and must reference where you got the code).
- Sound
- Loading of complex objects from files – if implemented assume the base of the object is bounded by an unseen cylinder so that your collision detection is still 2D circle-circle and 2D circle-line based
- Additional dynamic textures based on shaders.
- Geometry shaders. We will not be discussing geometry shaders in class so you would be on your own. Some geometry shading can be done in vertex shaders, however.
- Ability to hop over targets – must ensure you bounce off walls correctly. Note if you leave the XZ plane you must shift to 3D sphere-sphere and sphere-plane collision detection and collect significant bonus points.
- Add obstacles to the playing field in an appropriate way. Place a target in the center of a corn maze would be very Wisconsin and very mean. These might take the form of ramps if you implement true 3D motion or more simply “teleportation zones” which toss you someplace else in the arena.
- Instant replay from alternate camera angles with slow motion would be nice.
- If billiards action is implemented, properly rolling spherical targets (if not a spherical custom loaded object) will earn more bonus. Note that last year’s class implemented billiards so you might be able to hijack their code – but as stated – you will be responsible for their bugs and you must cite the source.
- Non-linear response to mouse position – in my reference implementation it is difficult to go exactly straight, exactly spin in place or stop. With a non-linear distance calculation you can make each of these capabilities easier / possible for the user to perform.
- Crowds and bleachers. In fact, a shader could be implemented that does the “wave” a “card dance” or other favorite crowd effects.
- Birds flying overhead either by flocking or faked with textures.
- Other atmospheric effects. A simple one would be just adding fog to the main display so you needed to navigate via the god-view (secondary viewport).
- Shadows. Especially if dynamically rendered and reactive to your movement (if you contain a light source). References are available on-line.

- Additional lights and light pylons. My reference implementation puts the light inside your sphere.
- Alternative camera views for the Jumbotrons. I originally had this in my reference implementation but, personally, it made game play harder because I couldn't reconcile what I was seeing on the Jumbotron with what I was seeing in the main screen.
- Debugging features such as enabling and disabling visible direct vectors.
- Multi-player – only for the very ambitious. Write networking code to connect more than one player together. Have one more target color than the number of players so that there is a neutral (un-bumped) color. Make the game time-limited and the winner is the person with the most targets his or her own color when time runs out. I can't help you with the networking code but I can tell you that you will need UDP and ports opened by the CSL.
- Real-time ray tracing – for the extremely ambitious. It is possible to do. You would be able to see yourself reflected in all the targets (and the targets reflecting each other).
- A very minor bonus and easy to implement would be a target visible only on the god-view but physically present in the arena. It would be an invisible target that you have to navigate into via instruments (the god-view).

Prohibitions:

- No shooting. Remember - Be the ball.

Game space

The arena is a square with 5280 units per side (think of this as feet, so a one square mile arena). Your player's radius is 50 (feet). The walls of the arena are double the height of the player's radius. Your player must be able to move swiftly across the arena as implemented in the reference implementation I provide.

The game should run only full screen mode (glutFullScreen()).

Shader requirements

All shaders must be implemented in GLSL. Alternative languages exist, but this is the one we use.

You must be able to demo in the CS labs. These machines are nVidia based. Therefore, using a GL extension found only on AMD cards will earn you a penalty since you won't be able to demonstrate them.

Walls and targets must both implement Phong (per pixel) shading. Other objects can as well.

Targets must implement at least one dynamically created (static or animated) texture or effect. A minimum of one more shader program must be implemented. More than one implemented additional means bonus.

That is, a minimum of three shader programs must be present in your application. A shader program is defined as a pairing of a vertex shader to a fragment shader. You are free to adapt shaders found on the Internet. Of course you are responsible for their bugs. **One emphasized requirement for shaders: if you find them with sparse comments, you must extensively comment them to prove to me you understand their method and function.**

Geometry shaders will not be discussed in this course. Use of a geometry shader is bonus.

Command line arguments

If one is provided, it must be the number of targets. If a second is provided it must be the seed for the random number generator. If the seed is missing, use zero so that you can demonstrate you get the same board as I render for the same number of targets.

Target placement

All implementations will seed their random number generators to 0 (or argv[2]) so that for the same number of targets, the layout is made. This will allow players to compare their performance with others (for bragging rights). Those implementing billiards-like action or true 3D movement will not be able to compare strictly apples to apples since every target strike will be calculated uniquely).

Usage of frame buffer object

At a minimum, you must render the main screen camera's view in a frame buffer object and use this frame buffer object as a texture to apply to "Jumbotrons" located on each side of the arena.

Floor effects

In my reference implementation I created a polished obsidian surface. You do not have to duplicate this effect. You <u>do</u> have to something cool on the floor.

Skybox or skydome

You must correctly implement a skybox or skydome. To receive full credit no seams can be visible and in the case of a skydome, no pinching can be visible.

Countdown timer

The value of the countdown timer (per struck target), must float above the target in an orientation that always faces the player's position. This calculation is trivial. If your method is seem difficult, there is a one line method.

The initial value of the countdown timer is computed with:

count_down_timer_seconds = 30 + number_of_balls * 2

Performance

Performance must be fluid and continuous.

Required key bindings

The ‘p’ key must completely pause the program and all related timers (this is can be more subtle than it sounds). The ‘w’ key must convert the rendering to wireframe. Remember, wireframe rendering can be accomplished in a single global state. Do not create separate objects for solid rendering and wireframe rendering. Function keys must toggle the use of each of your shaders in turn. The ‘x’ key and the escape key (decimal value 27) must exit the game prematurely.

Required on-screen text

These require text items must appear in a fixed position on-screen. The correct play time (accounting for pauses) must be shown as well as the number of targets remaining to be struck. If a target’s count down timer expires, the number of remaining targets increments. When a target is struck, decrement the number of remaining targets. The game ends when there are zero remaining targets (all targets are counting down). Choose a font which is located on all Windows machines. I chose Arial despite its plainness.

Required second view

As indicated in the reference implementation, there must be an overhead view provided (god-view). Your implementation specifics do not have to match mine.

Required command line output text

At the termination of the game, you must output the final time of termination (accounting for paused times) as well as an indication of whether or not you won.

Required libraries

You must use freeglut, DevIL, and some kind of extension “wrangler.” On Windows this is glew. If you add sound, a sound engine should be included. If you load objects from file, you will need code for that (of course).

No warnings

As in project 1, you will be asked to “build clean” then build. Any warnings (at the default setting) will result in a deduction.

Program exit

You should ask freeglut to continue running when glutLeaveMainLoop () is called. This will allow you to clean up memory but more importantly release resources living on the GPU.

Error messages and checking

You will be asked to start your program with various resources (textures and shaders for example) missing. Your program should handle this gracefully.

Use of code from other sources

You may use code from any source except:

1. From fellow students other than your partner.
2. Commercial sources (you cannot subcontract your class project).

You must cite your sources. As in project 1 you will be responsible for any bugs and warnings produced by lifted code. You must comment lifted code as if it were your own. You need to demonstrate you understand the code you borrowed.

Indicating that a target is struck

In addition to having a countdown floating above a struck target, the target itself must be rendered in a manner distinguishing it from targets that are not struck.

On-screen crosshairs

Aiming can be difficult; you must have a cross hair on-screen at all times to help you know where the center of the screen is.

Motion

You turn in the direction of the mouse relative to the center of the screen. Forward is up. Backward is down.

Bouncing

Hitting anything causes you to reflect in the appropriate direction.

Caution

Whether or not you implement billiards action, you must be prepared to discern if you are hitting a combination of objects within a single time quantum and handle the situation correctly. That is, breaking the time quantum down into a smaller unit and calculating exactly which object you hit first, calculating new positions, and completing the time quantum. This provides a hint as to an understandable algorithm for motion.

If billiards action is implemented remember to compute a rolling friction so that targets eventually come to a stop.

If billiards action is implemented note a key difference between this action and actual billiards. In actual billiards the cue ball receives only an initial impulse. In Moshball, the cue ball (you) has a motor. You may take the short cut of imagining you have infinite mass so a target striking you will bounce of you in an inelastic manner.

Screenshots

In your hand-in bundle please include a minimum of two screenshots of the coolness you have implemented. We will share these on a picture sharing service. **PLEASE include your name in the file name of the images.**

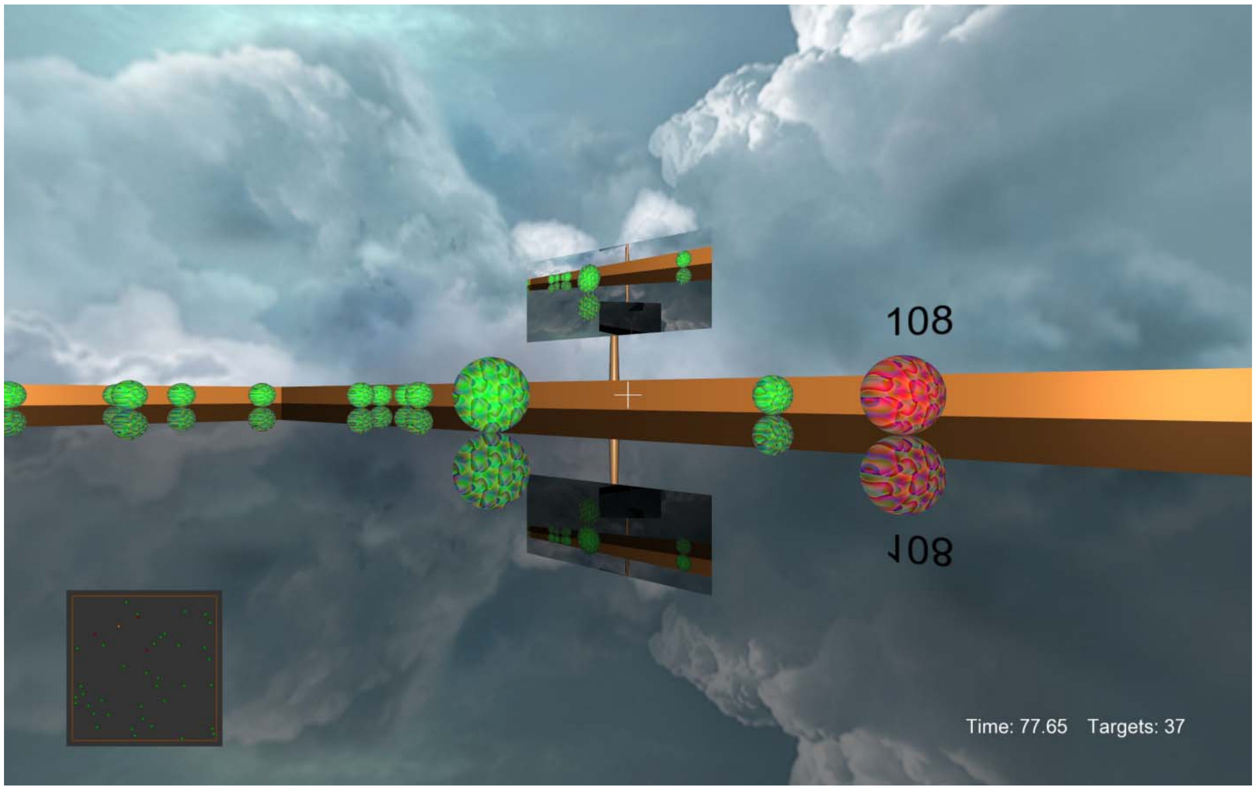
Expectation of the hours you will need

I required 20 hours to complete this project. I estimate you will need 80 to 100 depending on bonus features.

Class bake-off and competition

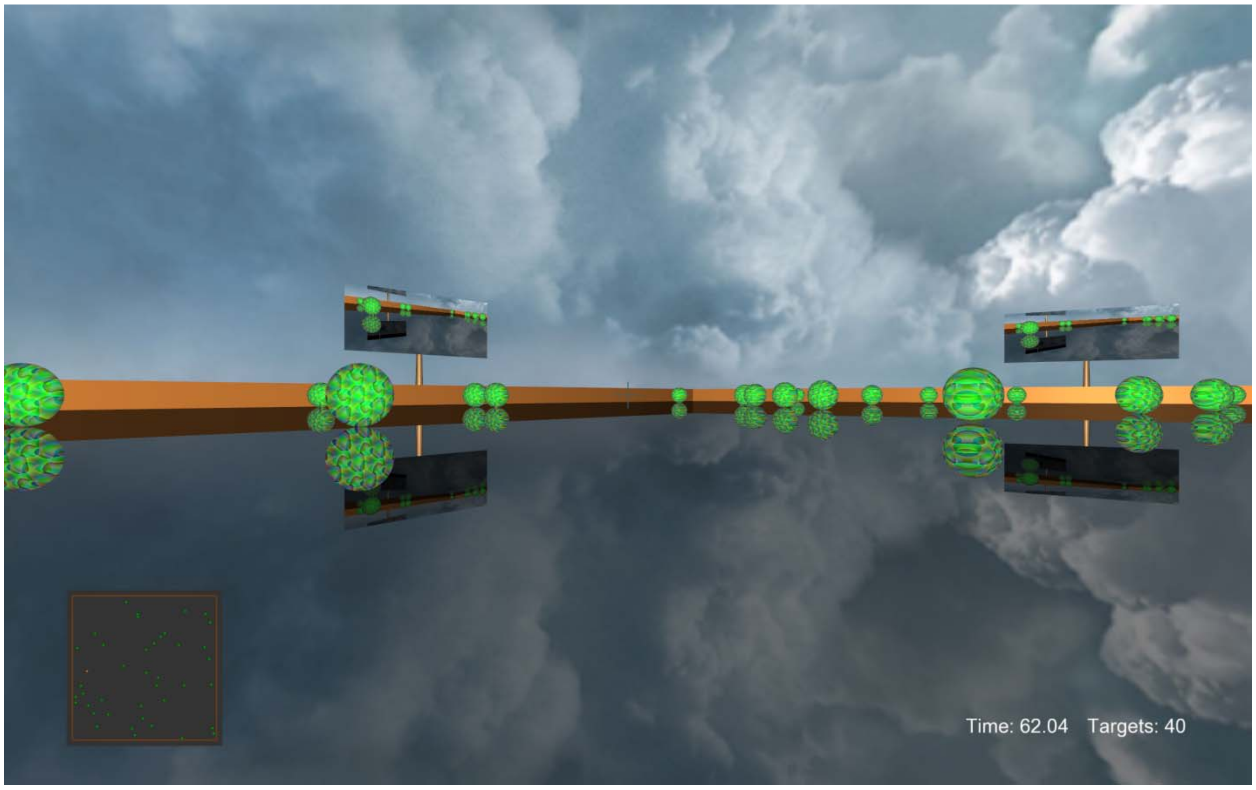
After the completion of the project, I will sponsor an evening of pizza and soda somewhere in the building. You bring capable laptops to show off your projects and compete head to head (where possible). You can bring a dish, dessert or beverage to share **but no alcohol or illegal substance is permitted**.

Example images



In this image you see the skybox I implemented. Note the presence and content of the on-screen text. Note the countdown timer floating above the struck ball (the red one). Further, note the master overhead view in the lower left. It must synchronize with the main screen including indicating which targets are struck (in this case by color). Finally note the floor effects I chose to implement – you may do something different but it must be dramatic. Notice the Jumbotron images which are placed in four locations. You may build a more elaborate Jumbotron for bonus.

Notice the dynamically generated texture on the targets. I chose to change this to a "screen" effect in the reference implementation.



Notice more than one Jumbotron.