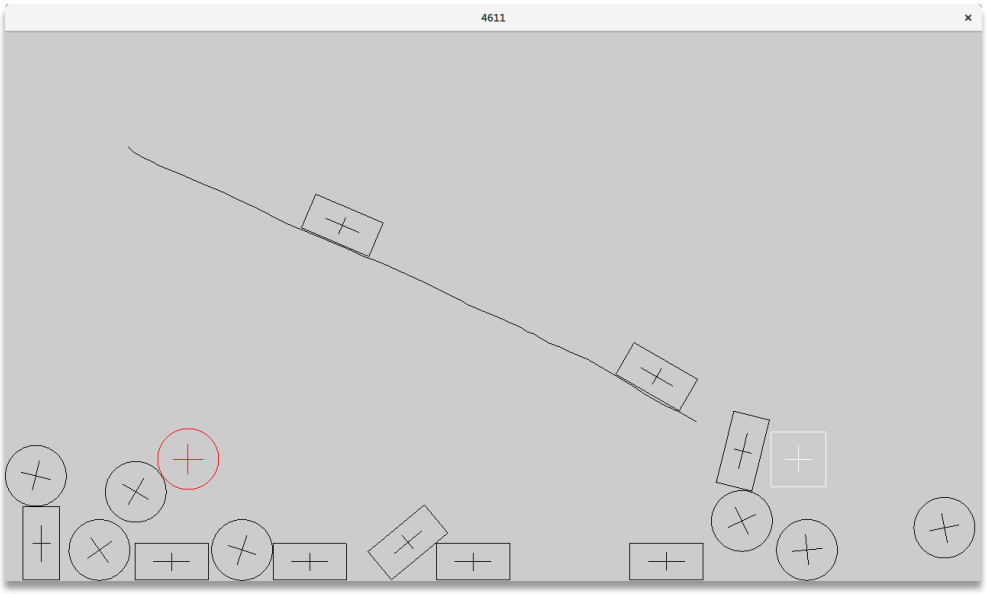
CSCI 4611: Programming Interactive Graphics and Games

Assignment 6: Pencil Physics

Handed out: Tuesday, April 18 (updated Thursday, April 20)

Due: Monday, May 1



Introduction

So far in the class, we've learned how to model, transform, and display shapes using a simple OpenGL-based toolkit. Game studios, movie studios, and others programming sophisticated interactive computer graphics applications regularly combine the graphics toolkits they use with other toolkits to support physics simulation, sound, and other effects. In this assignment, we'll do the same thing: you'll learn how to combine OpenGL with a physics engine, and in the process, you'll create a program that you should feel quite pleased showing off to friends and family!

Physics engines appear in graphics programs of all types, ranging from games such as Angry Birds to advanced modeling tools such as Maya. For simulating 3D physics, one of the most popular toolkits today is Bullet3D (http://bulletphysics.org/). In this assignment, we're going to simplify the physics a bit by limiting ourselves to physics within a plane (i.e., 2D physics), so we'll work with the Box2D physics toolkit, which is probably the current best / most popular 2D physics engine.

In this assignment, you will learn:

- how to create an interface between a graphics toolkit and another helpful library such as a physics engine,
- how to link physics with graphics, including dynamically creating and destroying physics objects and having them be reflected on the display, and
- how to interactively manipulate physics objects by translating user interaction into physical forces and/or constraints.

Requirements

For this assignment, you'll be hooking up the Box2D physics engine to the OpenGL-based starter code we provide for you. The aim of the assignment is to create a simple physics-based game which will drop circles and boxes on the scene, which the player must guide towards the white goal while avoiding the red circle. To do so, the user can draw curves on the scene to create static obstacles, and can also switch to a mode where they can move the circles and boxes around directly. The controls for the game are as follows:

- **B:** add box
- **C:** add circle
- **Tab:** switch from "draw" mode (where dragging creates obstacle curves) to "move" mode (where dragging moves objects around) and vice versa
- **Backspace:** reset scene

In the starter code, we have already implemented enough of the user interface code to allow you to draw curves and add circles and boxes. The user input handling is encapsulated in the `UIHelper` class, which calls various methods like `addCircle()` etc. on the main program depending on user input. We have also added a new `Draw` class that lets you draw simple shapes such as circles, boxes, and curves (actually polylines) under an arbitrary geometric transformation represented as a `mat4`. What remains for you to do is to hook up the physics simulation to the program to make the added circles and boxes move.

The specific requirements and suggested approach are as follows.

Link your program with Box2D

For this assignment, you'll be working with Box2D, so you need to change the way that you build your program to make it include the Box2D header files and link with the Box2D library file. This is similar to the examples we've done in class.

Set up the physics simulation

Your main task here is to figure out how communicate between Box2D and the shapes we've implemented already. For example, when a new circle is added to the scene, you'll need to add a corresponding object to your physics simulation. Then, you'll need to keep the physics and graphics in sync by advancing the physics simulation once each graphics frame, and

changing the way each shape is drawn to use the current pose of the corresponding physics object.

One way to do this is to change the definitions of all the shapes so that they keep track of a Box2D body associated with them. Then, when you want to draw a shape, you can look up the state of the body and use that as the transformation passed to the draw call.

Interaction

Using the provided interface code, the user should be able to create any number of different obstacle curves and dynamic circles and boxes in the world. You may assume that the user will not draw self-intersecting curves. Pressing Backspace should reset the simulation to the same state it was in when the program started, i.e. it should delete all the user-added objects in the scene.

The last thing you need to implement is support for moving objects around interactively. When the user presses Tab, the program switches to "move" mode, and UIHelper will call the methods `attachMouse()`, `moveMouse()`, and `detachMouse()` when the user presses down, drags the mouse, and releases the mouse respectively. To make the chosen object move, we will create a `b2MouseJoint` which acts as a spring pulling the object to a target location (in our case, the mouse position). So you will have to do the following:

1. Find the object that contains the world-space point where the mouse is, if any. To do so, update the `contains()` method of the shapes to account for the pose of the Box2D object. You may want to use the `b2Body::GetLocalPoint()` method for this purpose.
2. Create a mouse joint to the object, and set its target to the mouse location. Look up the documentation for joints in the Box2D manual to learn how to create a joint, and see the comments in the starter code for the parameters to use.
3. When the mouse is moved, update the joint's target using `SetTarget()` to the new mouse location.
4. When the mouse is released, destroy the joint.

Above and Beyond

All the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work — if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

Ideas for extra work include:

1. Detecting whether an object collides with the red circle or the white goal, deleting the colliding object, and showing a win/lose condition somehow.

2. Allowing the user to create dynamic circles and boxes at arbitrary positions and sizes by dragging the mouse like in a real drawing program.

3. Making more interesting levels, in the spirit of Crayon Physics or other physics-based games.

4. Improving the rendering to look (in some way) more impressive, such as drawing thicker lines and applying a crayon-like texture.

If your ideas for going beyond the requirements would make your code more difficult for the TAs to grade, please help them out by submitting a standard version of your assignment first through the normal website link, and then email the TAs the fancier version of your assignment.

Support Code

The webpage where you downloaded this assignment description also has a download link for support code to help you get started. The support code for this assignment is a simple program using the SDL-based engine, similar to the ones we have used before.

The support code defines a program structure and everything you need to read the mesh data. **To make locating the shaders simpler, we have a header file called `config.hpp` that contains absolute paths to your shader files. You should edit this file with the full path (e.g. `"C:\Users\Turing\a6\code"` or `"/home/turing/a6/code"`) where you've placed the shaders.** We will modify this file appropriately when grading your assignment.

Handing It In

When you submit your assignment, you should include a `README` file. This file should contain, at a minimum, your name and descriptions of design decisions you made while working on this project. If you attempted any "above and beyond" work, you should note that in this file and explain what you attempted.

When you have all your materials together, zip up the source files and the `README`, and upload the zip file to the assignment hand-in link on our Moodle site. Any late work should be handed in the same way, and points will be docked as described in our syllabus.