Introduction

You may have heard of the popular platformer game N; if you haven’t, you can play it at www.andkon.com/arcade/adventureaction/nthewayoftheninja/. In this assignment you will be making a simple platformer that implements a subset of the features of N, which is why we’re calling it M. You’ll add several important features to your engine, including some simple physics, raycasting, a system for loading levels from files, and an entity-I/O system for game logic.

Week 1 – Due Oct. 25, 2016

This week you will implement basic physics and collision response, and you’ll create a simple sandbox world to test out these features. Although compound shapes were an important part of Tou, you’ll leave them behind here because there is no simple method to compute collision responses for compound shapes. You will also update your collision debugger from Tou to show MTVs.

Design Check

- How will you ensure that the MTV you calculate is facing the correct direction?
- How will your physics engine be built into your existing engine structure? This includes detecting and resolving collisions, applying a correctional force or impulse, and generating a callback for your game code to make use of.
- How will you make static (immovable) objects static?
- How will you decide what correctional force/impulse to apply during collision?

Engine Requirements

- **Integration.** The engine must be able to update the velocities and positions of physical objects using symplectic Euler integration.
- **Force and Impulse.** The engine must allow the objects that it integrates to be affected by forces and impulses. These should accumulate during a timestep and be applied only at the end of the timestep, when integration is performed.
- **MTV.** The engine’s collision detection algorithms should find the MTV (Minimum Translation Vector) of a detected collision for all collision pairs except those that include compound shapes.
- **Collision Response.** The engine must provide a way to handle collisions between non-compound shapes by applying some kind of correction (impulse or force) in the direction of the MTV. In addition, the engine must allow game code to define how a particular collision is handled in addition to the correctional impulse or force.
Game Requirements

- The player must be able to control a character that can run left and right, and jump. The player should be affected by gravity.
- The world must be completely bounded by static shapes that do not move. No objects should be allowed to pass through these shapes.
- The world should have dynamic objects that are affected by gravity and collide with each other and the player. At least one of these dynamic objects must be less massive than the player and at least one must be more massive than the player.
- All of the shape types (except compound shapes) must be used at least once in collisions.
- The collision debugger is accessible and visualizes MTVs.

Extras

Getting your basic physics system in place may take some time, but if you want to keep building your engine, try the following!

- Support compound shape collisions.
- Have your viewport follow the player.

Week 2 – Due Nov. 1, 2016

Now you will improve your collision response to allow for inelastic collisions and provide the correct final velocity for the colliding objects. You’ll also implement raycasting, an important procedure for determining the effect of events in a game. To demonstrate these features you will add some simple weapons to your platformer, although you don’t have to keep them in the final version of your game.

Design Check

- Can you describe/explain both components of raycasting, meaning both the collision of a ray with a particular shape, and scanning all shapes to determine which was hit first?
- How will you implement grenades?
- How will you determine when the player is standing on a solid object?

Engine Requirements

- **Restitution.** The engine must support collision response using coefficients of restitution. It should combine the coefficients of restitution of two objects and use the formula given in class to accurately determine their post-collision velocities.
- **Raycasting.** The engine must support raycasting against circles, polygons, and AABs.
Game Requirements

- It must be possible to witness at least one elastic collision, one inelastic collision, and one perfectly inelastic collision.
- The player must be able to fire bullets/lasers that move instantly (using raycasting) and apply an impulse to whatever they hit in the direction of the ray.
- The player must be able to toss a grenade that explodes on contact with any surface and pushes all objects that are both within its radius and not obstructed by other objects.
- The player must be able to jump if and only if standing on a solid object.

Extras

You’ve got a nice physics system in place, so try to take advantage of it! Don’t forget about the M1 extras!

- Make your raycasting more efficient by avoiding iteration over all objects.
- Allow the player to “double jump” a small fixed number of times while in the air.

Week 3 – Due Nov. 8, 2016

In this week you will be adding an entity I/O system to your game engine. This is an important feature that allows game events to be defined in the level instead of hardcoded into the update loop. It is also now required to load your level information from a file, which will be created by a level editor that the TAs provide. To run the level editor, execute `cs195n_editor` from any directory. To get the support code that will assist you in reading the level editor’s files, copy the contents of the `m` directory in `/course/cs195n/support` to your project directory for M. This should add additional classes to the `cs195n` package in your `src` directory.

Design Check

- Can you describe/explain how Entity I/O works?
- Can you describe/explain how the level format we’ve given you works?
- How will you resolve entities’ className to a class?
- How will you resolve inputs’/outputs’ name to an Input or Output object?
- What are two standard entities you will make?
Engine Requirements

- **Entity I/O.** The engine must support the concept of an Entity with Inputs and Outputs, and must allow games to define connections between Entities.

- **Standard Entities.** The engine must provide a few standard, generic Entities that can be reused in any game, which will help make the entity I/O system easier to work with.
  - **Sensor** - upon collision with an object, sends an out-pulse
  - **Relay** - upon receiving an in-pulse, sends an out-pulse if open
  - **Other** - at least two other standard Entities (Timer, ForceField, Door, Filter are some examples)

- **Read Levels from Files.** The engine must be able to load LevelData objects from files created by the TA level editor, using the provided support code. It must be able to use LevelData objects to construct a World with Entities.

Game Requirements

- The camera must move so that the player is always in view.

- The game must construct its world from the level data provided by the TA level editor.

- The game must have non-arbitrary win and lose conditions.

- There must be at least one invisible Sensor entity that directly or indirectly causes a noticeable effect when triggered.

- There must be at least one Relay entity that can be easily enabled or disabled by player actions.

- Must make use of at least five Entity I/O connections

- You must implement at least four of the following features (note that some features may not make sense in the context of the game you are making, and that nested bullet points should only be done if their parents are done):
  - Multiple maps that automatically load and transition as the player continues through the game.
    - Create a mechanism for saving/loading progression through a series of levels
    - Make destructible environment, such as breakable blocks.
  - The game requires you to change gravity’s direction via some Entity I/O to complete it.
  - The player and enemies are drawn with sprites (and animations when appropriate) instead of vector graphics.
    - Use sprites for everything (no vector graphics).
  - A non-trivial puzzle that must be solved in order to win the game or complete a level.
  - Include multiple types of enemies.
    - Enemies that use complex behavior-tree AI instead of hardcoded behavior.
– Environmental traps, such as closing walls or collapsing ceilings.
– The ability to save and load games.
– A polished graphical UI system for non-game elements such as menus, with at minimum an options screen with at least two gameplay/control options
– A polished HUD with multiple buttons or controls.
– Any other feature(s) you can think of! Just approve it with a TA!

• It must be possible to start a new game without restarting the program.