CPSC 578 Final Project: Butter Lettuce

Marvin Qian

Concept

The game I created is a simple arcade game featuring Catbug, a character from the cartoon Bravest Warriors. The player controls one Catbug and there are five other Catbugs roaming the map. Each of the other Catbugs is holding a delicious head of butter lettuce and the player wants to catch all the other Catbugs and eat their butter lettuce. The bottom of the screen shows lettuce icons, one for each other Catbug, that will turn green one by one as the player eats the other Catbugs' butter lettuce. The player will have won the game once he or she has eaten all the butter lettuce and all the icons turn green.

Graphic Elements

There are four main graphic elements in the game: Catbugs, butter lettuce, rock formations, and flowers. The Catbugs and butter lettuce were modeled by manually specifying translations, scalings, and rotations of basic primitives. The rock formations and flowers were also done by hand but with a touch of randomness. The rock formations are built randomly level by level with the position of each level's rocks being based off of the average position of the previous level's rocks. The number of rocks has a random chance of decreasing as the levels increase, and the rock formation is complete once there are no rocks in a level. The flowers have random scalings, petal movement ranges, and petal movement speeds. All elements are placed at random within the map (except for the player, who starts in the center) and are ensured not to collide initially.

How to Play

To run the game, the player just needs to run a webserver within the main “butter-lettuce” folder and then navigate to the index page. For example, the player can run “python -m SimpleHTTPServer” in the “butter-lettuce” folder and then go to “localhost:8000” in his or her browser. Once the game loads, the player needs to click on the canvas in order for it to respond to keyboard input. Upon clicking on the canvas, the player is prompted to give permission for the page to lock the cursor. This is necessary in order to provide the camera view control that is expected from a third-person game. Once the pointer is locked, the camera view is pivoted around the player character according to the mouse position. To move around, the player uses the WASD keys (W moves forward, S moves backward, A moves left, and D moves right). The objective of the game is to collect all the other Catbugs’ butter lettuce by chasing them down and bumping into them. One can identify Catbugs that have already been caught as they will have stopped moving around the map and lost their butter lettuce. Once the player has collected all of the butter lettuce, the game is over and can be reset with a newly generated map by refreshing the page.

Challenging Elements

Animated Rigid Bodies

I spent many hours reorganizing the structure of entities within the NVMC framework. Instead of specifying the details for drawing each entity within the draw functions, the draw functions simply loop through an array of entities and calls draw on all of them. Each of these objects is a class that implements the draw function and has an instance of the Body class within them. The Body class represents a rigid body

that has a scene graph within it built of instances of the Node and Joint classes. Each Node can have a set of Primitive instances within them as well as a set of Joint instances. Each Joint instance has a child property that contains a child Node. The Primitive class represents a primitive shape and contains all of the information needed to render the primitive: the primitive's mesh, the shader to use, the color or texture of the primitive depending on which is applicable, and the Phong parameters (all objects use either the Phong shader that I modified to include shadows or the texture shadow shader that I modified to include Phong lighting). One specifies transitions, scalings, and rotations (specified as Euler angles since they are the most user-friendly) through an options object when instantiating instances of these classes. When one calls draw on the Body, it will call draw on the scene graph's root node which will then navigate through the whole graph of nodes, joints, and primitives to draw them all. When any node calls draw on a child node or primitive, it first applies its own relevant transformations so that they also influence its children and then pops them off the matrix stack before returning.

I also built a framework for animating these rigid bodies by rotating joints and modifying the root motion of the body through rotation or translation. Each instance of the Body class can take an array of animations within the options argument. An animation is a list of frames and a sequence that lists frame numbers and durations. Each frame is an object that specifies a state for the body (translation and rotation for root motion and rotations for all joints), while the sequence is an array of frame numbers (index into the array of frames) and corresponding durations in milliseconds. During animation, the current frame and the next frame in the sequence are interpolated to produce a state. Translations are linearly interpolated while rotations

are converted from Euler angles to quaternions, interpolated through quaternion SLERP, and then converted back to Euler angles. The root motion animation is intended to be relative motion (e.g., bouncing up and down). Global root motion is handled via user input for the player character and splines for the other Catbugs. The relevant code is in “game/body.js” and an example of a fully specified body with animations can be found in “models/catbug.js”.

Randomly Generated Spline Paths

In order to achieve smooth global root motion for the non-player characters, I implemented randomly generated spline paths. I used Catmull-Rom splines since they are interpolating and not approximating so I have more control over the path. Given an initial, randomly generated point as p1, the spline generator generates another nearby point for p0. The generator then selects all following p_i a certain distance away in a direction that is within 90 degrees of p_{i-1}-p_{i-2}. Since only four points are needed for the spline at any given time, new points are generated one at a time at a set interval that is checked on each frame. To sell the motion, I also modify the heading of the character to match the spline path. I do this by storing the previous position in order to calculate the tangent of the spline curve at each frame. Given the tangent, I can calculate the angle to the z-axis using the dot product as well as the direction to rotate by checking the sign of the cross product with the z-axis. The code can be found in “models/catbug.js” as it governs the motion of the non-player Catbugs.

Collision Detection

I also implemented a set of collision detection checks that all use axis-aligned bounding boxes. Collision detection splits updating object positions into a few steps.

The first step is to update the positions of every entity without considering collisions. Once these new positions are found, each entity updates its relevant AABBs. With the updated AABBs, the program checks collisions for each collider (entities that move around) against all collideables (static environment entities) as well as all other colliders. This is accomplished in three phases: 1) intersecting AABBs of the entire bodies, 2) intersecting AABBs of each primitive within the bodies, and 3) intersecting AABB bounding volume hierarchies of primitives within the bodies.

The first two phases are conservative checks that require preprocessing of the meshes to find the AABB vertices of the mesh in object space. On each frame, each primitive transforms its AABB vertices from object space to world space given its current state. This transformed AABB is no longer axis-aligned so the program has to find an AABB of the transformed vertices. This gives pretty awful bounding, but it only requires transforming eight vertices instead of all of the mesh's vertices. Given the vertices of the AABBs of all of the primitives within a body, we can also get an even more conservative AABB for the entire body which is used in the first phase. If a collision is found in the first phase, then the second phase checks the AABB of each primitive in the first body with the AABB of the entire second body. Each primitive of the first body that intersects the AABB of the entire second body is then checked against the AABB of each primitive in the second body. The resulting set of primitive pairs, for which a collision is detected in the second phase, moves onto the third phase.

The third phase is not an exact test, but a more accurate test that uses AABB bounding volume hierarchies. At this point, the program transforms all vertices of the primitive's mesh into world space. A bounding volume hierarchy for the transformed

mesh is then constructed top-down by splitting along the longest axis of the parent AABB at the mean position of its triangles until reaching a preset depth. The bounding volume hierarchies of the two primitives are then navigated one at a time, similar to how phase two first checks the primitives of one body against the entire other body before checking them against the primitives of the other body. If any of the leaves of the two bounding volume hierarchies intersect, then a collision is considered to be found and is appended to a list of collisions for that entity.

To handle collision response, I implemented a sliding mechanism that calculates the movement required for an entity to avoid a collision. I do this by finding the axis along which the AABBs intersect the least and moving the colliders that amount in the opposite direction. This is simple in the case of the AABBs that span an entire body, but it is made more complicated when using a bounding volume hierarchy. Since multiple pairs of AABBs can intersect within the two entities' bounding volume hierarchies, the program needs to take all of those intersections into account. If it did not, then the sliding displacement due to one intersected AABB could push the entity further into another intersected AABB resulting in strange behavior like phasing through rocks. To solve this, I track all the intersected leaves of a bounding volume hierarchy, and then find their lowest common ancestor which results in an AABB that contains all intersected AABBs. I then calculate sliding based off of these parent AABBs. A similar complication arises when an entity intersects multiple other entities. To incorporate sliding from multiple collisions, I take the maximum displacement along each axis. The end result is a robust collision detection system that only ever needs to calculate simple AABB to AABB intersections.

The player character handles collision response by adding the sliding adjustment to its new position whereas the spline-controlled characters reset their spline based off of their current position and the direction of the sliding adjustment. Collision detection is also used in initializing the game to ensure that entities do not start out colliding. The relevant code can be found in “game/collisions.js”.

Project Files and Organization

- game: contains game logic
    - events.js: handles user input and NVMC events
    - body.js: classes for animated rigid bodies
    - camera.js: specifies one third-person camera
    - collisions.js: handles building bounding volume hierarchies, finding AABBs, and checking collisions
    - draw.js: handles the per frame updates including updating entities and then drawing them
    - main.js: initializes shaders, loads textures, and sets game variables
    - normals.js: computes normals for triangles in a mesh
    - objects.js: initializes meshes, buffers, entities, and billboards
    - shadow.js: handles shadow map generation
    - skybox.js: handles sky box generation
    - textures.js: handles loading textures and rendering the shadow map texture
- lib: contains NVMC framework code
- media: contains textures, icons, and backgrounds

- models: contains meshes for primitives and entity classes
    - catbug.js: entity class and body specification for Catbugs
    - cone.js: cone primitive mesh
    - cube.js: cube primitive mesh
    - cylinder.js: cylinder primitive mesh
    - hill.js: entity class and body specification for rock formations
    - player.js: entity class for the player character and handles player movement
    - sphere.js: sphere primitive mesh with texture coordinates
    - texturedquadrilateral.js: quadrilateral primitive mesh with texture coordinates
    - tree.js: entity class and body specification for flowers
- shaders: contains shaders for rendering
    - billboard.js: shader for on screen billboards like the lettuce icons
    - phong.js: Phong lighting shader with shadows
    - shadow.js: shadow map shaders
    - skybox.js: shader for the skybox
    - texture.js: texture shader with shadows and Phong lighting
    - uniform.js: uniform color shader
- index.html & style.css: the actual web page and relevant styles

Compatibility

The game has been tested on Ubuntu 14.04 with Chromium 45 and Firefox 42.