Xtreme Fruit Simulator

COSC 435 - Graphics, Final Project Proposal

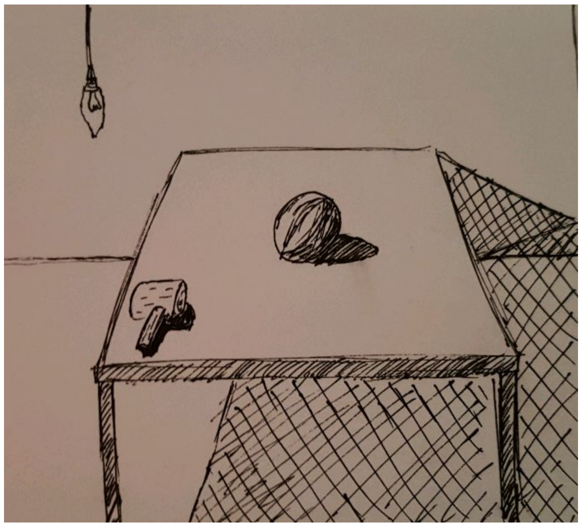
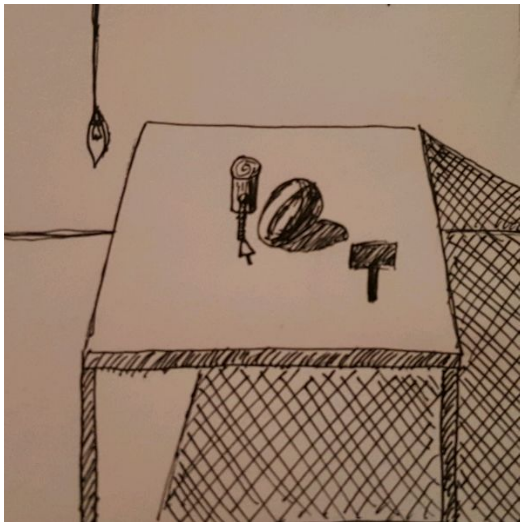
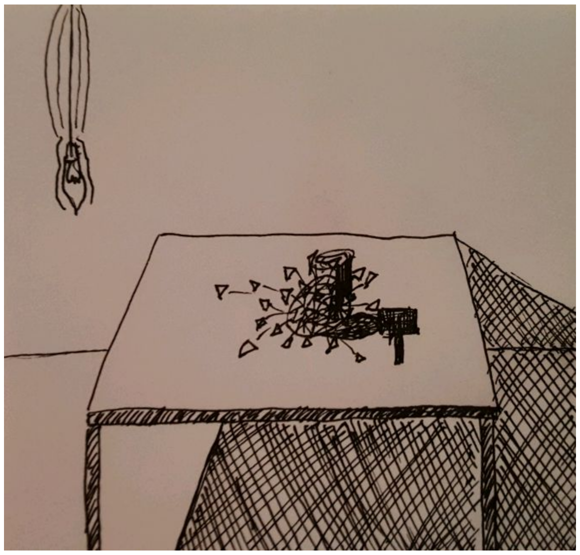
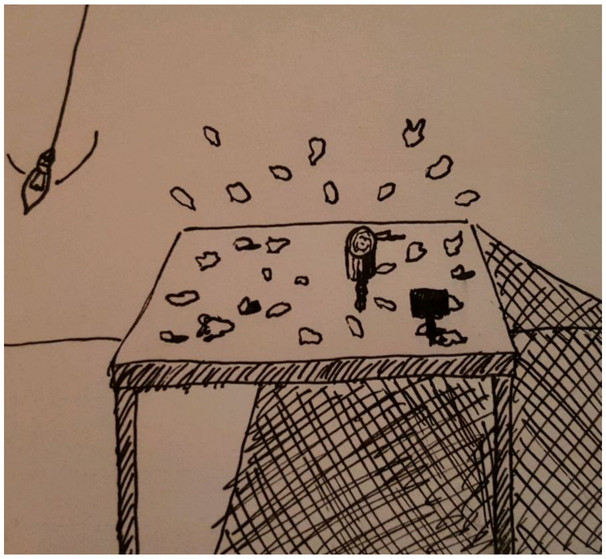
Logan Keig, Cory Nettnin, John Simoni

STATEMENT

Our goal is to create an interactive sandbox-style game in which the user is able to interact with objects. We want to create a sort of narrative, not through the use of text or dialogue but, instead, through the use of aesthetics and player action–we want to show, not tell.

The scene depicts a somewhat dimly-lit basement environment, with the focus on a table pushed up against a wall. Near the table is a hanging light. On the table is a hammer and a piece of fruit. Gnats buzz around the rotting fruit to complete the dingy aesthetic.

The key user action in the game is picking up the hammer and smashing the fruit, with other smaller interactions available throughout the scene by "using" other objects in the environment (such as the fruit "wiggling" and turning the light on and off). The focus of the project is to display the effects of a fruit explosion once it has been hit by the hammer. We will be showing the fruit breaking apart into fruit fragments. These fruit pieces will fly throughout the scene and collide with the wall, table, and any other objects on the table.

TECHNICAL OUTLINE

- Exploding fruit
  - Predefined
    - Use knife cuts in blender to create new sets of vertices, and then use the separation tool to split that set of vertices into its own object.
    - A single predefined sectioning of the fruit will be what we use to begin with, and then, given enough time will move to either a selection of several pre-defined cuts or something that randomizes.
- Physics engine
  - Physijs - has objects that take the place of THREE.js meshes and scene
    - Physijs.PlaneMesh - infinite zero-thickness plane
    - Physijs.BoxMesh - matches THREE.CubeGeometry
    - Physijs.SphereMesh - matches THREE.SphereGeometry
    - Physijs.CylinderMesh - matches THREE.CylinderGeometry
    - Physijs.ConeMesh - matches THREE.CylinderGeometry (tapered)
    - Physijs.CapsuleMesh - matches THREE.CylinderGeometry, except has two half spheres at ends
    - Physijs.ConvexMesh - matches any convex geometry you have
    - Physijs.ConcaveMesh - matches any concave geometry you have, i.e. arbitrary mesh
      - According to the github, ConcaveMesh has the worst performance
      - This is a potential hurdle, and may prove to be our bottleneck in limiting the number of objects in the scene at any one time.
    - Physijs.HeightfieldMesh - matches a regular grid of height values given in the z-coordinates
  - Call scene.simulate() in the render function to perform calculations
  - We can attach collision listeners to objects in order to trigger events such as fruit explosions, splatter, and hammer marks on the table.
    - var mesh = new Physijs.BoxMesh(geometry, material);
      mesh.addEventListener('collision', function( ) { ...});
  - The ceiling light could be implemented using a hinge constraint, an object that constrains the motion of another as if it were on a hinge.

    -
      ```
      var constraint = new Physijs.HingeConstraint(
         physijs_mesh_a, // First object to be constrained
         physijs_mesh_b, // OPTIONAL second object - if omitted then
      physijs_mesh_1 will be constrained to the scene
         new THREE.Vector3( 0, 10, 0 ), // point in the scene to apply the
      constraint
         new THREE.Vector3( 1, 0, 0 ) // Axis along which the hinge lies - in this
      case it is the X axis
      );
      scene.addConstraint( constraint );
      constraint.setLimits(
      ```

```
    low, // minimum angle of motion, in radians
    high, // maximum angle of motion, in radians
    bias_factor, // applied as a factor to constraint error
    relaxation_factor, // controls bounce at limit (0.0 == no bounce)
);
constraint.enableAngularMotor( target_velocity, acceration_force );
constraint.disableMotor();
```

- Decals
    - Basic Decal Algorithm
        - Run a raytrace, checking for intersections with other objects.
        - Assuming there is an intersection, save the point of intersection
            - Use lookAt function to obtain a vec4
            - Use extractRotation to determine the angle of the splatter on the surface
        - Using the position and rotation, create a new decal object
    - In our case, the raytrace is less essential, and the point of intersection will be determined instead by points of impact.
        - These points can be saved in a queue, and created during calls to the function responsible for adding objects.
- Bump mapping
    - Basic algorithm
        - Convert the texture to grayscale
        - Use this grayscale image to represent a height map
            - The brightness of each pixel represents how far it stands out from the surface
        - Create a vector for each pixel, representing the “downhill” direction
            - x_gradient = pixel(x-1, y) - pixel(x+1, y)
            - y_gradient = pixel(x, y-1) - pixel(x, y+1)
        - Adjust the normal vector of the polygon at that point
            - New_Normal = Normal + (U * x_gradient) + (V * y_gradient)
        - Calculate new brightness at that point using the adjusted normal
    - Our application will vary slightly in that we plan to construct our bump map off of an existing 3D object. Additionally, we can complete all of these steps from within Blender.
        - Use the Render Bake feature to obtain Bump and/or Normal maps
        - Place the map on a low-poly model that uses the same coordinate system as the original, high-poly, model
- Picking
    - Using a function onDocumentMouseMove, we would keep track of the location of the mouse in the canvas, and, if an object is “selected,” the object will be moved along with the mouse location.

  - With functions onDocumentMouseDown and onDocumentMouseUp, we would detect mouse clicks and, if the location of the mouse is the same as the location of one of our pickable objects, we would mark the object as “selected.”
  - Initially the picking will be hard coded to keyboard keys. We will most likely use the suggestion of “H” for the hammer until we get the mouse coding correct.
- Shadowmap
  - Adding the shadows is relatively straightforward: a light source (specifically, a spotlight) should be added wherever our physical hanging light is found, that moves whenever the light itself moves. The light, table, fruit, and hammer need to have “castShadow” set to “True” and the floor and background wall need to have “receiveShadow” set to “True.”
  - The more difficult part of the shadowmaps is removing the shadows when the light source is turned off. Due to the way that graphics are drawn, the shadows remain even if the light source is off (as they’ve already been drawn). In order to fix this, when the light source is turned off, the floor and wall materials need to be updated by setting the “material.needsUpdate” to “True.”
  - This was implemented in Cory’s HW2 as the optional objective, and will be implemented in a similar manner in this project.

SOURCES

Link To Tutorial/Code Used	**Objective Code Was Used For**
https://www.blender.org/manual/modeling/meshes/editing/subdividing/knife_subdivide.html	1 (Modeling)
https://www.blender.org/manual/modeling/meshes/editing/subdividing/knife_subdivide.html	1 (Modeling)
https://www.blender.org/api/blender_python_api_2_77_0/info_quickstart.html	1 (Modeling)
Lab 8 (JSON exporting and importing)	1 (Modeling)
http://www.textures.com	2 (Textures) & 9 (Decals)
http://stemkoski.github.io/Three.js/Textures.html	2 (Textures)
http://stemkoski.github.io/Three.js/Texture-Repeat.html	2 (Textures)
github.com/mrdoob/three.js/blob/master/examples/webgl_materials_bumpmap.html	3 (Bump mapping)
freespace.virgin.net/hugo.elias/graphics/x_polybm.htm	3 (Bump mapping)
https://aerotwist.com/tutorials/creating-particles-with-three-js/	4 (Particles)
github.com/mrdoob/three.js/blob/master/examples/webgl_lights_physical.html	5 (shadowmap)
http://stemkoski.github.io/Three.js/Shadow.html	5 (shadowmap)
github.com/mrdoob/three.js/blob/master/examples/webgl_interactive_draggablecubes.html	6 (pickable objects)
Jenga example from Physijs code	6 (pickable objects)
github.com/chandlerprall/Physijs	7 (physics engine)
https://github.com/chandlerprall/Physijs/wiki/	7 (physics engine)
https://github.com/chandlerprall/Physijs/wiki/Updating-an-object's-position-&-rotation	8 (animation)
github.com/mrdoob/three.js/blob/master/examples/webgl_decals.html	9 (decals)
https://github.com/spite/THREE.DecalGeometry	9 (decals)
http://stemkoski.github.io/Three.js/Shadow.html	10 (GUI)
http://stemkoski.github.io/Three.js/Color-Explorer.html	10 (GUI)

OBJECTIVES

1. Modeling of four key objects using Blender: the table, the lightbulb, the hammer, and the fruit. We will also be modifying our fruit model in Blender to form the pieces that appear when the fruit is broken. The dimensions of each individual fruit will also be randomized using basic scaling transformations.
2. Textures will be applied to all objects to make them look more realistic.
3. Bump mapping will be used on the surface of the fruit and the table to give the appearance of a realistic surface (of varying heights) instead of just a flat texture.
4. A particle effect will be used to show “gnats” swarming around the fruit before the fruit is hit with the hammer.
5. A shadow map will be implemented. Shadows will be generated from the single hanging bulb, and the other three main objects (the table, the fruit, and the hammer) will all cast shadows.
6. All objects (excluding the table and light) will be pickable. All objects (excluding the table) will be “usable.” The light will toggle off and on. The fruit will play a small jiggle/shake animation. The hammer will allow the player to hit the fruit or the table.
7. A physics engine will be implemented to handle object collision and gravity. This is essential for the fruit explosion and trajectory of the resulting pieces.
8. Animation will be added to various small parts of the game. The fruit will “jiggle” when “used”, the hammer will hit, and the light will sway when the table is hit by the hammer.
9. Two types of decals will be added to the game. When the hammer misses the fruit and hits the table, a decal of a dent will be applied to the table. When the fruit pieces collide with objects (either the wall, the table, or other pieces of fruit) a splatter decal will be applied at the location.
10. A menu will be implemented through a GUI that will contain options to spawn more fruit, the power bar that determines how hard the hammer hits the fruit, and will also have a reset button to return everything to the original layout (where the hammer is in its starting position, the splatter and table dent decals are all cleared, and a single piece of fruit is on the table).

OBJECTIVES ANALYSIS

1. We successfully modeled our four objects in Blender. We used Knifing to cut the fruit object into multiple pieces for the explosion. We were unable to vary the sizes of the models. Exporting and importing several sized models created too large of an overhead, especially since the pieces for all of those differently sized models would have also needed to be imported. Resizing the models within the Javascript itself led to collision detection issues within the physics engine, specifically the Javascript resized models would only collide based on their original size.
2. Textures were successfully applied to most objects. The fruit texture would only apply to each of the individual faces on the fruit, so it came out looking a bit odd, but still gives relatively the same appearance we were looking for.
3. In addition to applying bump mapping to the fruit and the table, we also applied bump mapping to the walls.
4. Particle effects have been added to simulate gnats swarming around the fruit. The particles float above each fruit object until the fruit is smashed. The number of gnats is randomized per fruit.
5. Shadows were successfully implemented. The shadows also correctly move when the light source moves, which was not part of our original plan but was easy enough to add in at the same time. The table, fruit, and hammer all both cast and receive shadows.
6. In addition to making the fruit and hammer pickable, we also made the light pickable. The light is usable: the L key will toggle the light on and off.
7. The physics engine was successfully implemented. The gravity is used to propel the hammer towards the fruit. Collision detection is implemented in a couple of different places: the hammer detects when it collides with the table (to apply a dent decal) or a piece of fruit (to break it) and the subsequent fruit pieces from breaking a piece of fruit all have collision detection to detect when they hit a non-table object. Pieces that land on the table are left intentionally unbroken, as these would be the pieces that had the least amount of force applied to them in a real life situation.
8. Animation was added to the light bulb. When the table is hit with the hammer, the light will shake a bit. Animation was unable to be added to the hammer because of a conflict with the physics engine that made repositioning outside of the physics engine nearly impossible.
9. We were unable to get the decals to work properly even after several iterations of code. We were able to load the decals in and detect when they should be applied, but the decals themselves would never show on the objects they were being placed on. The code has been left in, but commented out, in the event you want to read through it.
10. A menu with a power slider and buttons to spawn fruit and reset the layout was added. The power slider is moved manually with the mouse and the fruit requires a certain power level in order to break. The button to spawn fruit will spawn one fruit per click and has no limit as to how much it can spawn. Each spawned fruit is identified by a number, with the original fruit object being “fruit0”. The reset button removes all added objects besides the table, walls, ceiling, and floor, and then re-adds the lightbulb, hammer, and a single piece of fruit (“fruit0”) to the scene in their original positions.

Minimum Viable Product (MVP) Development Plan

- Phase 1: *The Scene*
    - Cory: Shadows
    - Logan: Models
    - John: Physics
    - **4/11 - Connect** into full, non-interactive scene with lighting, shadows, and physics
- Phase 2: *User Interaction*
    - Cory: UI, Spawning, Reset
    - Logan: Picking
    - John: Blender Knifing and fragmented models
    - **4/18 - Connect** into full, interactive scene with spawnable fruit and movable objects
- Phase 3: *The Climax*
    - Cory: Decals
    - Logan: Animation and the “usability” of objects
    - John: Fruitsplosion
    - **4/25 - Connect** into full, interactive scene with our main attraction: fruitsplosion!
- Phase 4: *Aesthetics*
    - Cory: Particle effects
    - Logan: Textures and bump mapping
    - John: Quality Assurance and Stretch Goals
    - **5/2 - Connect** into a final product