Physics API

ODE

The Open Dynamics Engine (ODE) is a rigidbody physics simulation engine. A rigidbody object is represented dynamically by its location, velocity, rotation, and angular velocity, and statically by its mass, center of mass, and inertia. The rigidbody is actually not defined by its shape, you only need a shape to represent the objects geometry. A physics object's geometry is the shape used during collision detections, while its body is what is used to interact with and respond to forces in the physics world. An object can have either a body, a geometry, or both, depending on how you want the object to act. A typical simulation loop using ODE will look something like this:

- Create a dynamics world.
- Create bodies in that world.
- Set the position, rotation, and other state information of those bodies.
- Create joints in the world and attach them to bodies.
- Set the parameters of all the joints.
- Create a collision world and the geometries for objects as needed.
- Create a joint group to hold the contact joints (the temporary joints created when a collision happens).
- In the event loop (called with render() in our graphics engine, however it could be called less often since physics calculations tend to be expensive):
  - Apply forces to bodies as needed.
  - Adjust joint parameters as needed.
  - Call collision detection to check for collisions.
  - For every contact point, create a contact joint and put it in the contact group.
  - Step the simulation forward by some amount of time (recommended to be constant, and not large).
  - Remove all of the joints from the contact group because they are temporary since the represent collisions.
- When everything is finished, destroy both the dynamic and the collision world.

API

Important methods:

- `pInitShape()`: Used to create the desired shape in the physics simulation. Can be made whatever size and shape the user wants. The physics body and/or geometry does not need to match the graphical object. Currently we only support 4 primitive objects: sphere, box, capsule, and cylinder.
- `pSetTranslation(), pSetRotation(), pGetTranslation(), pGetRotation()`: These allow the user to get and set the position and rotation of the associated physics body in the physics simulation. Then they can apply the result to their objects however they choose.
- `pInitJoint()`: Let's the user specify what sort of joint they want as well as which two objects that joint should connect. Specific parameters for each joint can be set later with various setter functions. Currently we support 4 joint types: fixed, ball, hinge, and slider (although fixed joints should only be used for debugging). The joints can also be enabled and disabled.
- `pSetForce(), pSetTorque(), pGetForce(), pGetTorque()`: These allow the user to apply force/torque to specific objects. Can either be called as a one-shot, or in the simulation loop to apply a consistent force to the object.

Scene Graph

We decided that the Physics API should be completely abstracted from a scene graph, since you do not need a scene graph to implement a physics engine. Then, we had to decide how to integrate our API calls into our scene graph structure:

1. The scene node object now stores three additional pieces of information: a `physicsLoc`, which is an integer that represents the given node's physics body and/or geometry in the physics simulation, a `jointLoc`, which stores the integer representing the joint connected to this physics object in the physics simulation (this could be extended to include multiple joints), and `worldPos`, which is the world position of the node after being transformed by all its parents. It is used to "compromise control" over the scene node if it has a parent(s) and an associated physics body in the simulation.
2. Rather than completely ignoring the scene graph structure if an object has a physics body, the parent node has "semi-control" over the child. The calculation used to do this is as follows for some frame:
    - Let $\vec{p_0}$ be the location of the physics body in world coordinates from the previous frame. Let $\vec{p}$ be the location of the physics body in the world of this frame.
    - Let $\vec{t}$ and $\vec{w}$ be the local translation and world position of the scene node respectively.

- Then, the new translation of the scene node should be set to: $\vec{t} - (\vec{p_0} - \vec{p})$. This is essentially allowing the regular scene graph translation to happen, and tossing the physics movement on top of it.
- The position of the physics body should be set to $\vec{w} - (\vec{p_0} - \vec{p})$ ($\vec{p_0}$ should also be set to this for the next frame).
- Note, there's probably a similar compromise to be made for rotation, but we did not implement one.

The decision to allow "semi-control" over physics bodies in the scene graph came from the Unity Game Engine. When a rigidbody is a child of another object that isn't a rigibody, the rigidbody will sort of follow its parent around while still being subject to physics, which is what the above decisions are attempting to replicate. This structure will achieve somewhat realistic results. However, this does mean that when you have a rigidbody as a child of another rigidbody, there will be some very unrealistic behavior. This is because there are too many different transformations acting on a given scene node. There's the physics forces acting on both objects, as well as the local transtions within the scene graph. In fact, the Unity API states that you should never do this, because it will not produce realistic behavior. When parenting a rigidbody to another rigidbody, what you probably want is some sort of joint or composite object instead.