Game Physics
Assignment Sheet 2

Prof. Dr. Jan Bender, MSc. Andreas Longva

Contact: {bender, longva}@cs.rwth-aachen.de

SS 23
Deadline: **04/05/23**

The purpose of this assignment sheet is twofold. On the one hand, you will get more experience with and be acquainted with the Godot game engine. On the other hand, you will get some practical experience with some basic physics simulation concepts that we will generalize and extend in later assignment sheets.

In the following exercises, you will be setting up multiple scenes with different purposes. This fits in quite nicely with Godot, as it already lets you create several scenes within a single project and run them individually.

To solve the exercises, you are expected to use online resources to learn the necessary details. In particular, you must learn to use `GDScript`, the scripting language in Godot, and in general actively consult online documentation.

Remember that you should hand in the source code of your work, including any Godot project files. The material you provide must be self-contained, in the sense that all the solutions to the exercises should be runnable.

Additionally, your group should present their findings in a presentation in the first meeting after the deadline. The presentation should be no longer than 10 minutes. Please prepare this presentation well, so that you may present your results in a structured and timely manner.

Assignment 1 - Camera

In this first exercise, we will build a simple, interactive Camera, which we can reuse in various assignments to follow.

Godot's *Camera* node does not come with any interactive controls, so we will have to provide this ourselves. Thankfully, this is quite straightforward. There are presumably many ways to accomplish this. Here we use a simple approach to build a camera that works in the following way:

- Pressing e.g. keys *A, D, W* or *S translates* the camera left, right, forward or back with respect to the current transform of the camera, respectively.
- Pressing e.g. keys *Up, Down rotates* the camera up, down with respect to the current transform of the camera, respectively.
- Pressing e.g. keys *Left, Right rotates* the camera left or right with respect to *the global up vector*, respectively. Using a fixed up vector instead of the up vector corresponding to the current transform of the camera is usually more intuitive for camera controls, since it gives the user a sense of what is up and down.
- The speed of the translation and rotation is constant independently of frame rate.
- (Optional) Mouse look for controlling orientation.

Just like a rigid body, the camera has its own transform, which defines how its own local coordinate system is transformed into the global coordinate system. By convention, the camera defines *forward* as pointing in the local negative z direction, *right* as the local positive x direction and *up* as the local positive y direction.

a) Create a simple test scene with some objects and set up a Camera node.

Recall from the *Getting Started* documentation that you may attach scripts to virtually any node. For our camera, we will attach a new script and override the `_process(delta)` function. Refer to the documentation for the Input singleton. In particular, `is_key_pressed` might be useful.

b) Implement the desired functionality by changing the transform of the Camera depending on which keys are currently pressed. How can you ensure that the camera moves at a constant speed when keys are held down?

c) Make the Camera with its attached a script into an instanced scene by right-clicking your Camera node and selecting *Save Branch as Scene*. This way you can easily add the camera functionality to other scenes by right-clicking your root node and choosing *Instance Child Scene*.

Assignment 2 - Particle motion

In this exercise, we will manually perform time integration of a particle (which we represent as a sphere) and perform very simple collision detection and response against a planar surface. The techniques we use here are very simplified and specialized to our particular problem, but some of the same ideas are used in more advanced algorithms that deal with e.g. multiple simultaneous collisions and constraints on a system of bodies.

Note that we will completely bypass the physics engine for this exercise, as we will not use any 'RigidBody' nodes or similar.

a) Create a new scene with a directional light and a camera. Create a new *Spatial* node. This will serve as our "particle". Add a *MeshInstance* node with a spherical shape. Set the radius to a value of your choice.

The *Spatial* node in Godot only has spatial "position-like" properties. That is, it has a *transform*. It has no concept of velocity, so we will have to introduce this ourselves.

b) Attach a script to the *Spatial* node and add member variables for velocity and gravity (both 3-dimensional vectors). Note that you can use the `export` keyword to make the properties editable in the Godot Inspector.

   Implement the *semi-implicit Euler* method to integrate velocities and positions when the particle is subject to gravity. You may implement the physics logic in the method `_physics_process(delta)`, where `delta` corresponds to the time step. Verify that your particle moves as expected when influenced by gravity. In particular, choose an initial velocity such that the particle trajectory is a parabola.

   Hint: To update the position, it suffices to update the "transform.origin" property with the new position value that you have computed.

Next, we will allow the spherical particle to collide with a "floor", represented as a plane. Our strategy is simple: if the sphere penetrates the half-plane defined by $y \leq 0$, then move the sphere such that it exactly rests on the plane.

c) Add a large floor located exactly at $y = 0$ to the scene. Implement the collision detection and response as described, by modifying your code for the time integration to check whether the spherical particle is in contact with the plane. What do you do with the velocity?

So far we disallow the spherical particle to penetrate the planar floor, but it directly stops as it hits the floor. For a more lively animation, we want it instead to bounce when it hits the floor. For this purpose, we employ Newton's law of restitution:

$$v_{\text{rel}}^{+} = -e\, v_{\text{rel}}^{-}.$$

The above relation dictates that when a collision occurs, the *relative velocity* $v_{\text{rel}} := \mathbf{n}^T(\mathbf{v}_1 - \mathbf{v}_2)$ measured along the contact normal $\mathbf{n}$ after an impact (indicated by $+$) must be be proportional to the negative relative velocity before the impact (indicated by $-$). $\mathbf{v}_1$ and $\mathbf{v}_2$ represent the velocity of the contact points at each of the two bodies in contact, and e is a constant that determines the amount of "bounce", and is typically in the interval $[0, 1]$. Note that here we neglect the effect of rotations, otherwise the relative velocity v_{rel} must include additional terms.

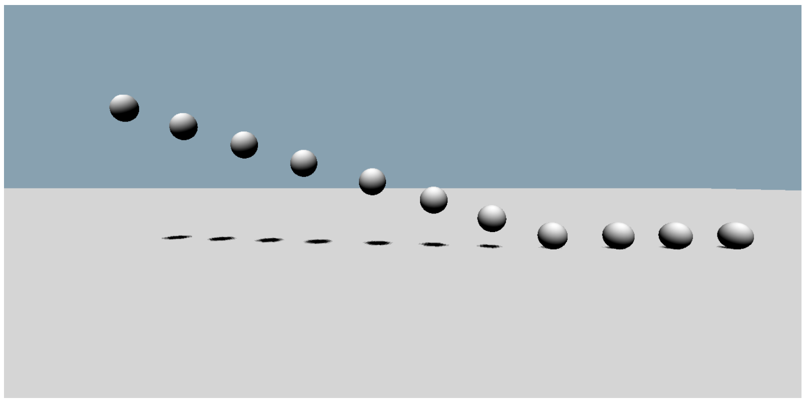

Figure 1: A series of spherical bodies with different coefficients of restitution.

d) Modify the collision response with the plane such that the velocity is modified instantaneously according to Newton's law of restitution. Make `restitution` a configurable property of your object. Set up multiple spherical particles next to each other with different restiution values and comment on the results. See figure 1 for how the result might look like.

   Hint: Since the plane is static, you can set $\mathbf{v}_2 = 0$, and consider $\mathbf{n}$ to be the vector perpendicular to the plane (the surface normal of the plane).

e) (Optional) Add additional constraints to prevent particles from penetrating walls defined by $x = \pm b$, $z = \pm b$, for $b > 0$.

Assignment 3 - Spherically constrained motion

Consider a rigid body with position $\mathbf{X}$. Our goal is to constrain the motion of the rigid body to the surface of a sphere with center $\mathbf{x}_0$ and radius r. That is, we wish to ensure that $\mathbf{X}$ is always a point on the sphere. Figure 2 demonstrates how this may look like.

As in the previous exercise, we will develop a specialized procedure to achieve our goal. However, unlike the method we developed there, we will this time integrate with the physics engine present in Godot.

Whereas the function `_physics_process` runs before every tick of the physics engine for any object, the function `_integrate_forces(state)` runs only for rigid bodies, and it is the preferred place to e.g. add forces, impulses or even directly adjust positions or velocities.

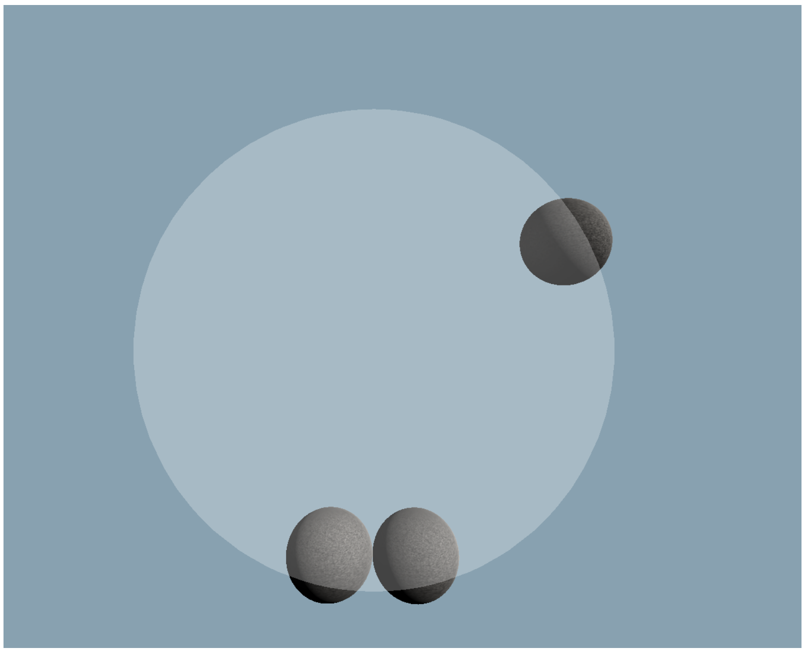

Figure 2: Spherical bodies constrained to move on the surface of a sphere.

Our strategy is simple: At each time step, move the body to the closest point on the sphere. This is the same as *projecting* the current position $\mathbf{X}$ onto the sphere.

a) Create a new scene with a *RigidBody* node. Attach a script, and override the function `_integrate_forces(state)`. The `state` variable is an instance of PhysicsDirectBodyState. For this exercise, you mostly only need to access the `transform` property of the `state`.

   Devise and implement a method to project the current position $\mathbf{X}$ onto the sphere of center $\mathbf{x}_0$ and radius r. In addition, find a way to visualize the sphere that the body is constrained to move on. Test your method with multiple initial positions for the body. Explain and demonstrate your approach.

Hint: Draw the problem in 2D on a piece of paper. The extension to 3D is straightforward.

b) After implementing the position projection, you may notice that the body slowly drifts away from the sphere. Can you explain why?

In addition to the drift problem alluded to above, the body does not behave quite naturally: due to gravity, the body directly moves to the bottom of the sphere and stays there. We would perhaps expect it to also move sideways within the sphere, and not immediately settle in the bottom. Otherwise the motion is not energy conserving.

To solve the aforementioned problems, we must also adjust the velocity of the body. To see how, assume that the body is already on the sphere. For the body to stay on the sphere, its velocity must be tangent to the surface, otherwise it will immediately move away from the sphere. Our strategy is as follows: After the position correction, remove the part of the velocity $\mathbf{v}$ that is perpendicular to the sphere's surface.

c) Implement the proposed method. Explain how you accomplished the velocity correction. Verify that the body stays on the sphere without drift, and that its movement is more consistent with what we would expect from energy conservation.

   Note: Due to the linear/angular "damping" introduced by the physics engine, your body will quickly reduce its energy. You can reduce/disable this by changing these properties directly for a body, or globally for the physics engine in the project settings.

d) Add more bodies and corresponding collision shapes, and verify that your spherical constraint also works together with the collision response given by the physics engine.

Assignment 4 - Ball joints

In this exercise, we will acquaint ourselves with arguably the simplest type of mechanical joints: the ball joint. Refer to the crash course notes for a discussion of ball joints. Recall that a ball joint between two rigid bodies is defined by two local attachment points $\mathbf{r}_1$ and $\mathbf{r}_2$, defined in the local coordinate systems of body 1 and 2, respectively. The ball joint enforces the requirement that the *world coordinates* of the two attachment points remain equal. Mathematically, we write

$$C_B(\mathbf{z}_1, \mathbf{z}_2) = \mathbf{r}_1^w - \mathbf{r}_2^w = 0.$$

For the time being, we eschew the precise definitions of $\mathbf{r}_1^w$ and $\mathbf{r}_2^w$ and refer to the crash course notes.

In Godot, ball joints are represented by *Pin Joints*. The setup is a little different than what we just described. To use a Pin Joint in Godot, you set up your rigid bodies in the way you want them attached (in world coordinates), and then add a PinJoint to the world position of the common attachment point.

To relate the two representations, consider the fact that when the joint is satisfied, $\mathbf{r}_1^w = \mathbf{r}_2^w =: \mathbf{r}^w$. Godot's Pin Joint thus lets you directly set the initial world coordinate $\mathbf{r}^w$, and it internally computes the corresponding local attachment points $\mathbf{r}_1$ and $\mathbf{r}_2$. This is arguably a natural way to work with the joint in the editor, as it lets you set up your rigid bodies directly in a valid state, and simply place the joints directly at the attachment points.

a) Create a new scene. Place one rigid body without a shape exactly in the origin, and set its *Mode* to *Static*. Place another (dynamic) rigid body with e.g. a spherical shape exactly in the same location, and add a *Pin Joint* node to the origin as well. Attach the two bodies with the pin joint by setting the *Nodes* properties of the pin joint.

   Verify that when you run the simulation, the dynamic body stays pinned to the static body in the origin.

b) Add several other identical bodies connected by pin joints in a horizontal line. See Figure 3. Verify that the bodies fall with gravity, but remain connected through the joints.

   Hint: You may want to add textures to your bodies, so that you can more easily observe rotation for e.g. spherical objects. For example, you can use automatically generated noise textures.

c) A sequence of bodies connected by ball joints is probably the easiest way to simulate a chain or rope-like structure. Attach more bodies to the joint and let them interact with other bodies in the scene. For example, you can simulate a chain falling onto a floor (or something more complicated) this way.

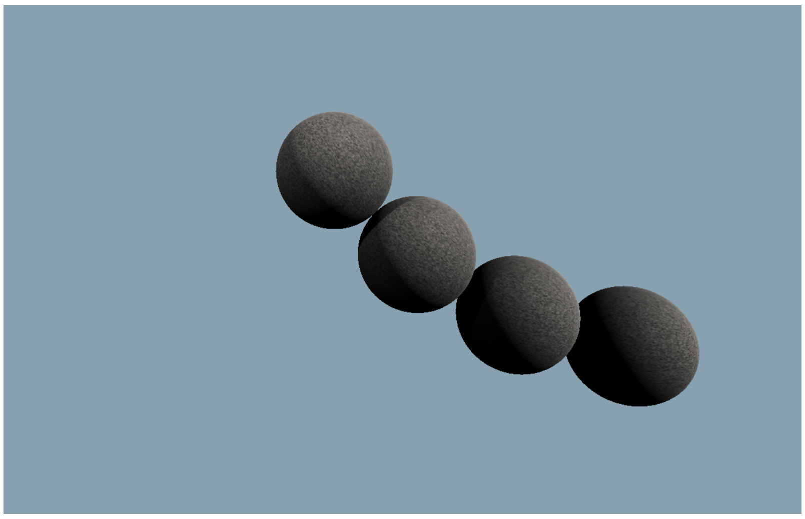

Figure 3: A series of spherical bodies connected by pin joints.