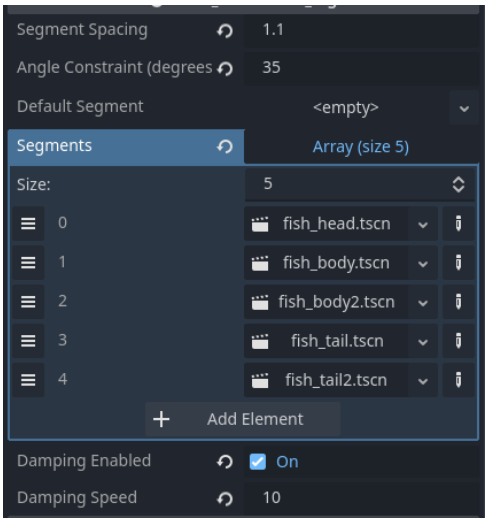
Team AKIK Final Project Report

Overview:

Our final project is a GDExtension Procedural Animation Tool that allows developers to simplify the process of creating and prototyping procedurally animated entities without having to implement procedural animation themselves. This was accomplished by creating custom nodes that each encapsulate a different procedural animation technique. The two main techniques used are inverse kinematics (IK) and distance constraints. The nodes are compatible with each other, allowing designers to combine different procedural animation techniques to create unique and complex animations.

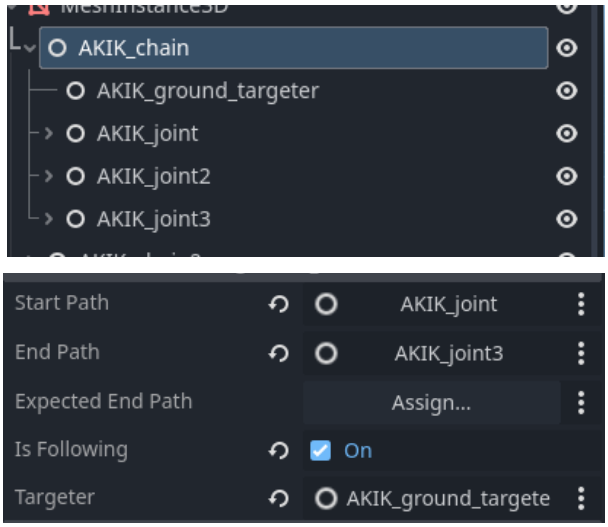
Software Architecture / Design:

- **AKIK_Procedural_Rig Class**
    - This node allows the user to create a procedurally animated entity by defining a chain of scenes
    - The node uses distance constraints as its main procedural animation technique. To use it the user must define an array of scenes. The node is animated both in the editor and at run-time. The node also offers a rich set of parameters for designers to quickly make modifications to the node.
        - Segment Spacing
            - Defines the distance between segments
        - Angle Constraint (degrees)
            - Limits the angle of bend between two segments. Values are in units of degrees and are limited between values of 0 - 360.
        - Default Segment
            - A scene object which will be replicated for each new segment added to the segments array.
        - Segments
            - An array of scenes that define what scene will be instantiated at each segment. Each segment is fully customizable, as the user can drag and drop any scene into any segment of their choosing.
            - Null/empty elements are allowed and will create an empty segment as a placeholder
        - Damping Enabled
            - If enabled, the rig will straighten over a period of time when at rest, eventually arranging the segments into a uniform line.
        - Damping Speed
            - The speed at which the chain will straighten itself




```cpp
void AKIK_Procedural_Rig::_process(double delta)
{

    float max_angle = Math::deg_to_rad(angle_constraint);

    for (int i = 0; i < chain.size() - 1; ++i) {
        Node3D* node = chain[i];
        Node3D* next_node = chain[i + 1];

        Vector3 diff = next_node->get_global_position() - node->get_global_position();
        Vector3 constrained_offset = diff.normalized() * segment_spacing;

        Vector3 current_dir = diff.normalized();
        Vector3 prev_dir = (i > 0) ? (node->get_global_position() - chain[i - 1]->get_global_position()).normalized() : Vector3(0, 0, -1);

        // Apply damping
        Vector3 damped_dir = current_dir;
        if (i > 0) {
            if (damping_enabled) {
                Vector3 ideal_dir = (prev_dir + current_dir).normalized();
                damped_dir = current_dir.lerp(ideal_dir, delta * damping_speed);
            }

            float angle = damped_dir.angle_to(prev_dir);
            if (angle > max_angle) {
                Vector3 axis = prev_dir.cross(damped_dir).normalized();
                Transform3D rotation_transform = get_transform().rotated(axis, max_angle);
                damped_dir = rotation_transform.basis.xform(prev_dir).normalized();
            }
        }

        // Move next node to stay within constrained distance
        constrained_offset = damped_dir * segment_spacing;
        next_node->set_global_position(node->get_global_position() + constrained_offset);
        node->look_at(next_node->get_global_position(), Vector3(0, 1, 0));
    }
}
```

- **AKIK_Chain Class**
    - This acts as the overall manager for a single IK chain. Allows users to create a chain of nodes that procedurally follows a set position. This can be used to simulate limbs of an animal.
    - This node is animated both in the editor and at run-time, allowing the user to prototype from the editor and preview its procedurally animated movement without having to use the play-in-editor.
    - To create a chain, the user will add a *AKIK_Ground_Targeter* node and any number of *AKIK_Joint* as a child of a node of this type. References to these nodes will need to be set within the node properties.
        - Start Path
            - Node reference to first joint in the chain
        - End Path
            - Node reference to last joint in the chain
        - Is Following
            - Begins procedurally animating for this chain. Enable this after chain and joints have been fully configured
        - Targeter
            - Node reference to targeter node




```cpp
void AKIK_chain::fabrik_step() {
    float total_distance = 0.0f;
    AKIK_joint* curr = start;
    while (curr->next != nullptr) {
        total_distance += curr->get_next_length();
        curr = curr->next;
    }

    if (total_distance <= start->get_global_position().distance_to(expected_end_node->get_global_position())) {  // outsi
        curr = start->next;
        Vector3 oldPos = start->get_global_position();
        while (curr != nullptr) {
            float distance = curr->get_prev_length();
            Vector3 newCurrPos = (expected_end_node->get_global_position() - oldPos).normalized() * distance + oldPos;
            oldPos = curr->get_global_position();
            curr->set_global_position(newCurrPos);
            curr = curr->next;
        }
    } else { // within range
        float tolerance = 0.01;
        if (end->get_global_position().distance_to(expected_end_node->get_global_position()) > tolerance) {
            start_anchor = start->get_global_position();
            forward_kinematic();
            backward_kinematic();
        }
    }
}
```

- **AKIK_Joint Class**
    - Stores its position and connection to other nodes. Keeps track of the length between its adjacent nodes. Each joint can be visualized by attaching a mesh as a child of the node.
        - Next Path
            - Node reference to the next joint following the current join. Empty/null if last join in the chain.
        - Prev Path
            - Node reference to the previous joint preceding the current join. Empty/null if firstjoin in the chain.

```cpp
void AKIK_joint::_ready() {
    this->next = (this->next_path == NodePath("")) ? nullptr : this->get_node<AKIK_joint>(next_path);
    this->prev = (this->prev_path == NodePath("")) ? nullptr : this->get_node<AKIK_joint>(prev_path);

    this->next_length = (this->next == nullptr) ? 0.0f : get_position().distance_to(this->next->get_position());
}
```

- **AKIK_Ground_Targeter Class**
    - Performs a raycast to check for a surface to move the target to during runtime and in-editor.
        - Length
            - Determines the length of the surface detection raycast. Ideally, this length is set to a value similar to the length of the chain.
        - Jump Margin
            - This value determines how far the end of the chain and deviates from the position of the target before it is reset to the target's position.

```
void AKIK_ground_targeter::_process(double p_delta) {
    if (target == nullptr) {
        return;
    }

    force_raycast_update();
    if (is_colliding() && is_jump_margin_traveled()) {
        target->set_global_position(get_collision_point());
        //instead of instant set, need to draw a parabola arc peaking at a height above where raycast hits.
    }
}
```

- **AKIK_Follower Class**
    - This node allows for custom position interpolation. This can be used to add extra movements to a node when moving within the world, creating the illusion of weight and acceleration and giving more variety and realism to the entities movement.
        - Target_Path
            - Node reference to the node that will follow this node.
        - Frequency
            - How quickly the entity will follow this target.
        - Damping
            - Determines the stiffness of the entity upon reaching its target. Lower damping means the entity will bounce back and forth around its target eventually coming to a stop.
        - Initial
            - Determines rate of acceleration of the entity towards the target.

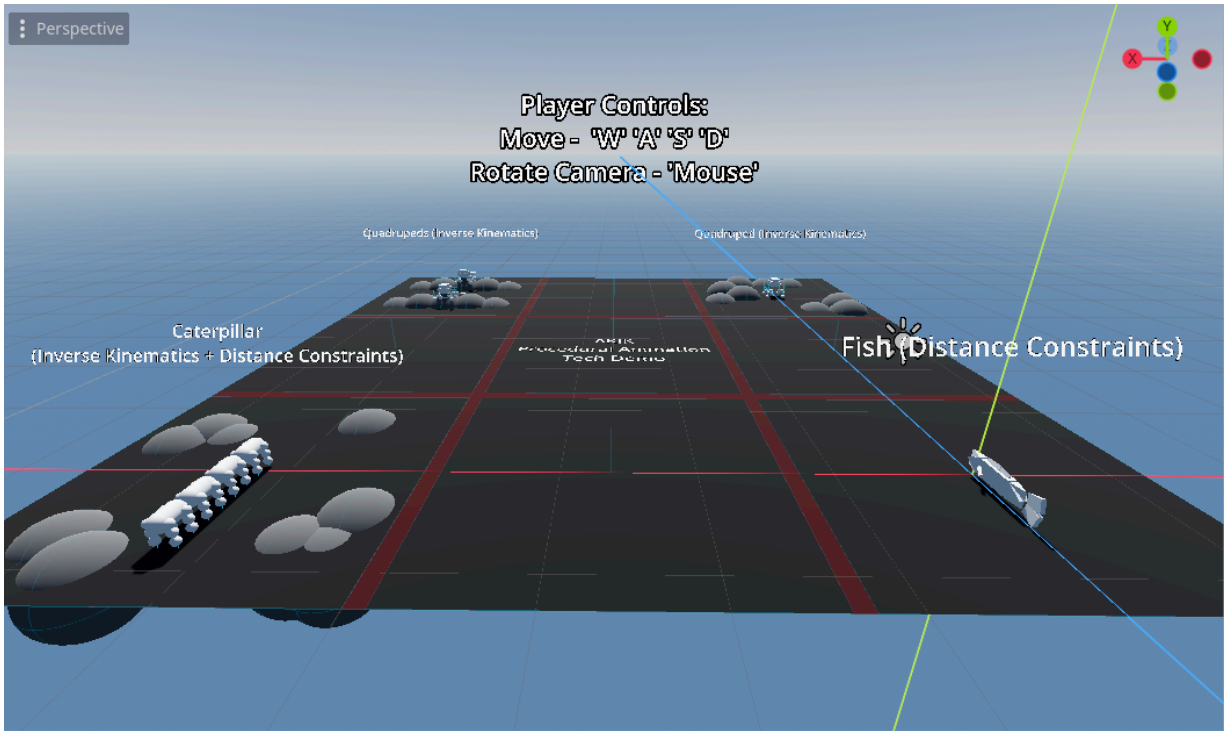
```
void AKIK_follower::_process(double p_delta) {
    if (target == nullptr) {
        return;
    }
    Vector3 x = target->get_global_position();
    Vector3 xd = (x - xp) / p_delta;
    xp = x;
    y = y + p_delta * yd;
    yd = yd + p_delta * (x + k3*xd - y - k1*yd) / k2;
    set_global_position(y);
}

void AKIK_follower::compute_constants() {
    k1 = z / (Math_PI * f);
    k2 = 1 / ((2 * Math_PI * f) * (2 * Math_PI * f));
    k3 = r * z / (2 * Math_PI * f);

    xp = (this->target == nullptr) ? get_global_position() : this->target->get_global_position();
    y = xp;
    yd = Vector3(0, 0, 0);
}
```

- **Demo Scripts**
    - We have created a variety of C# and gdscript scripts to provide basic movement to our nodes for the sake of demonstrating our procedural animation nodes in our technical demo. Most of these scripts are some variation of interpolating the node's position from some combination of a sin and cos wave function.
    - We also have a player script that provides basic player movement such as WASD controls and third person movement via mouse.

```
1 reference
public override void _Ready()
{
    // Store the initial position of the node
    startPosition = Position;
    randomOffsetX = GD.Randf() * Mathf.Pi * 2; // Random value between 0 and 2π
    randomOffsetZ = GD.Randf() * Mathf.Pi * 2; // Random value between 0 and 2π
}

1 reference
public override void _Process(double delta)
{
    // Calculate the new X and Z positions for the squiggly movement
    float x = startPosition.X + Mathf.Sin(Time.GetTicksMsec() / 1000.0f * speed + randomOffsetX) * amplitude;
    float z = startPosition.Z + Mathf.Cos(Time.GetTicksMsec() / 1000.0f * frequency + randomOffsetZ) * amplitude;

    // Set the node's position, maintaining the initial Y position
    GlobalTransform = new Transform3D(GlobalTransform.Basis, new Vector3(x, startPosition.Y, z));
}
```

```
func _physics_process(_delta):
    # Get the input direction and handle the movement/deceleration.
    # As good practice, you should replace UI actions with custom gameplay actions.
    var input_dir = Input.get_vector("left", "right", "up", "down")
    var direction = (transform.basis * Vector3(-input_dir.x, 0, -input_dir.y)).norma
    if direction:
        velocity.x = direction.x * SPEED
        if (!Input.is_action_pressed('up')):
            velocity.y = 0
        else:
            velocity.y = -pivot.rotation.x * SPEED

        velocity.z = direction.z * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0, SPEED)
        velocity.y = move_toward(velocity.y, 0, SPEED)
        velocity.z = move_toward(velocity.z, 0, SPEED)
    move_and_slide()
```

- **Tech Demo Scene**
    - Our tech demo scene is an environment that helps demonstrate our custom nodes. There are a variety of groups of entities present in the scene. The user also plays the demo as a dragon which was animated using our custom nodes. The scene is divided into four sections, each highlighting a different procedural animation technique. There is also varied terrain for some sections. Sections are labeled with billboarded Label3D nodes for clarity.
        - Dragon Player - Animated with AKIK_Procedural_Rig
        - Fish - Animated with AKIK_Procedural_Rig
        - Quadruped - Animated with FABRIK nodes (Chain, Joint, Targeter)
        - Caterpillar - Animated with FABRIK + AKIK_Procedural_Rig



Challenges:

- There was added difficulty in working with nodes within the editor compared to only using them in run-time. Oftentimes the editor would crash from a segfault, making it difficult to find the source of the bug as errors couldn't be printed into the console.
- Lack of documentation forced us to pivot from our original idea. Originally, we wanted editor buttons to be able to allow for tool configuration, however, there is little to no online resources for creating editor buttons through GDExtention
- Time limitations due to holiday break and busy schedules. We would have preferred to have spent more time on the project to further polish our node functionalities, but we lacked the time due to obligations for other classes/work.

Successes:

- We were able to accomplish our goal of making a generalizable procedural animation tool. We were even able to make our nodes compatible with each other, allowing for more versatility in how they are used.
- Figuring out how to animate nodes within the editor was very cool to see once we got it working. This project has definitely taught us more about GDExtension than any of the past projects.
- We also learned how to design tools while considering the needs of designers. Overall very satisfied with the quality of tools we were able to make.