IMG-420 Final Project: Deliverable 1

Procedural Animation and Distortion GDExtension

Course: IMG-420: 2D Game Engines
Due Date: Tuesday, November 18th 2025
Team Members: Cole Bishop, Tyler Jeffrey, John Zeledon

---

1. Executive Summary

Project Title: Slime Chronicles

Team Members and Roles:

Name	Role	Responsibilities
Cole Bishop	Systems Developer	Core C++ GDExtension implementation, Gameplay Mechanics outlining, performance optimization.
Tyler Jeffrey	Gameplay Programmer	Integration of procedural animation system into gameplay, collision response, and slime behavior.
John Zeledon	Designer / Technical Artist	Parameter tuning, visual style development, test scenes, user interface setup.

Overview:

This project aims to implement a procedural animation and distortion GDExtention for Godot 2D in C++. The goal of this extension is to enable real time mesh deformation and procedural motion in 2D sprites and meshes. Rather than relying solely on shaders or pre-baked animations this system provides physics based and noise based deformation that is directly accessible from scripts and the godot editor. To showcase this extension, it will be powering a demo game called “Slime Chronicles”, a 2D platformer where the player controls a jelly like slime creature that dynamically stretches, squashes, and ripples in response to motion and impact.

The main goal of the **DistortionSprite2D** extension is to fill a gap in Godot’s current 2d pipeline, which lacks direct support for vertex-level deformation. While shaders can approximate soft-body effects, they cannot interact with the physics system or individual sprite vertices. By implementing the deformation logic in C++, this module provides fine-grained control over vertex positions, UV coordinates, and per-frame transformations, making a short game prototype

“Slime Platformer” where both the player and enemies behave like soft, elastic creatures. When landing, they compress and rebound; when dashing, they stretch or ripple outward; and when idle, they subtly oscillate due to noise-driven motion. This game will serve both as a performance testbed and a creative proof of concept.

Developing the module poses several technical challenges. Integrating deformation into Godot’s 2D rendering pipeline requires manipulating mesh data efficiently without breaking batching or UV mapping. The physics-based model must remain numerically stable using mass-spring or position-based dynamics while running at high frame rates. Noise-based deformation must generate smooth, time-dependent vertex motion using algorithms such as Perlin or Simplex noise, optimized to handle hundreds of vertices per frame. Another challenge involves maintaining performance updating vertex buffers without excessive CPU–GPU communication and ensuring stable collisions even as visuals deform. The team must also balance visual realism with performance by supporting modes for visual-only deformation or approximate collider adjustments.

Other complexities include designing an intuitive editor experience, allowing developers to fine-tune elasticity, damping, amplitude, and noise parameters through live previews and preset systems. Compatibility across platforms, efficient memory use, and deterministic physics behavior are also priorities. Finally, the module will feature seamless TileMap integration, handling per-tile deformation without visible seams. The completed deliverables will include the C++ GDExtension library, inspector UI controls, preset profiles, a playable slime demo, and a performance report. The project’s success will be measured by stable, smooth animation at 60

Key Technical Challenges:

- Implementing real time vertex deformation on CPU side meshes within Godots 2D renderer
- Designing a performance conscious system that updates deformations every frame without breaking frame pacing.
- Building a flexible C# API layer for artists and developers to adjust physical system parameters easily.
- Synchronizing sprite deformation logic with physics events like collisions, jumps, and attacks.
- Creating robust and comprehensive editor integration for real time tweaking and visualization.

2. Module/Extension Specification

Overview

The Procedural Animation & Distortion GDExtension is a C++-based Godot GDExtension that introduces a new node type: **DistortionSprite2D**. This node is a deformable 2D mesh capable of simulating soft body or noise-driven motion. It functions similarly to Sprite2D, but instead of a static rectangular mesh, it maintains a custom mesh whose vertices are updated each frame.

Per Frame Vertice Updating is Handled Through:

1. Physics Based Spring Lattice (Enables Jelly Like Deformation)
2. Noise Field Oscillation (Handles Ripples and Idle Wobble)
3. Collision Impulses (Creates Reactive Squash and Stretch Effects)

This GDExtension exposes all parameters to the Inspector and provides a simple C# API for controlling deformation programmatically.

Functionality Summary

Feature	Description
Procedural deformation	Real time mesh deformation using spring damper physics
Noise distortion	Perlin/Simplex noise to create natural idle wobble or rippling (Continuous Noise functions that generate natural looking random variation)
Impulse response	Collision impacts propagate vertex displacements
Custom node type	ProceduralSprite2D extends Node2D
Editor parameters	Elasticity, damping, noise amplitude, frequency, and more
C# API access	Methods to trigger impulses, set deformation states, or freeze motion

Technical Architecture

1. Core Class **DistortionSprite2D**:
   - Manages vertex data (ArrayMesh)
   - Computes Deformation using spring physics and noise functions
   - Updates the mesh surface each frame
2. Helper Classes:
   - **Deformation Field**: This class handles generating the procedural deformation values (How much to move each vertex and in what direction). It can get deformation input from noise functions, impacts, and other in game events.
   - **ElasticBody**: Maintains the vertex positions, velocities, and spring forces. The Deformation field defines the noise fields and other impacts imposing force onto the elastic body while the elastic body handles how the vertices of the body react to force.
3. Design Patterns:
   - **Component Pattern:** Multiple Nodes make for modular integration of the system
   - **Observer Pattern:** Events in the world like collisions are listened for so they can always be reacted to dynamically.
   - **Factory Pattern:** Procedural meshes are generated upon deformation. Instead of modeling each object we are generating a dynamically deformable mesh grid.

API Documentation

Class: DistortionSprite2D

Method	Description
void apply_impulse(Vector2 point, Vector2 force)	Applies an external deformation impulse
void set_elasticity(float elasticity)	Sets the spring return strength
void set_damping(float damping)	Sets energy loss per frame

void set_noise_strength(float noise)	Controls amplitude of procedural noise
void set_texture(Ref<Texture2D> tex)	Assigns texture to mesh surface
void reset_mesh()	Restores mesh to original shape

Editor Properties:

- **Elasticity(float)**: Controls how strongly the sprite's vertices try to return to their rest positions after being displaced. How "bouncy" or "rubbery" your distortion is. Higher value is more rigid, snaps back to shape quicker and tighter.
- **Damping(float)**: Controls how much motion is smoothed out over time by reducing oscillations. It's the "energy loss" factor of your jelly simulation. Higher value means motion settles quickly, minimal bounce and vibration.
- **Noise_amplitude(float)**: Defines the strength of the procedural noise distortion applied by the DeformationField. This controls how far each vertex moves from its base position due to the Perlin/Simplex noise field. High value is strong exaggerated wobbling and ripples.
- **Noise_Speed(float)**: Controls the rate of change of the procedural noise field over time. How fast the ripples move or the idle wobble animates. High value has rapid jittery distortion.
- **Subdivision(int):** Defines how many segments or grid divisions the deformable mesh uses. Higher subdivision = more vertices = smoother, more detailed distortion. This can and will affect performance, higher value is a larger load on processing.
- **Texture(Texture2D):** The base image applied to the deformable sprite mesh. This is what the player actually sees. As the vertices distort, the texture's UVs deform accordingly, creating the illusion that the image itself is stretching or rippling.

Integration with Godot

- Compiled as a GDExtension dynamic library
- Loaded automatically via res://addons/procedural_animation/
- Register class using godot::ClassDB::register_class<DistortionSprite2D>();
- Accessible in C# via standard GD.Load() and new ProceduralSprite2D() calls

- Meshes and parameters are visible and editable in the Godot Inspector

---

Comparison to Existing Solutions

Method	Pros	Cons
Shader-based deformation	Fast, GPU-accelerated	Hard to sync with gameplay physics, not CPU-accessible
Animation frames	Easy to create	Not dynamic or reactive
Our GDExtension	Procedural, dynamic, and physics aware	Slight CPU cost, limited by mesh complexity

---

3. Game Design Document

Concept

Title: Slime Chronicles
Genre: 2D Physics Platformer
Platform: PC / Windows
Engine: Godot 4.5

You play as a sentient slime that can bounce, stretch, wobble, and shoot slime balls through puzzle-like levels filled with enemies, traps, and interactive soft terrain.

Core Gameplay Mechanics

- **Movement**: Jump, Slide left and right, and dash.
- **Scaling**: Get bigger or smaller with the press of a button.
- **Combat**: Shoot slime balls at enemies to take them down.
- **Deformation**: Player Character squishes and stretches based on external forces.
- **Environmental Interactions**: Jelly Platforms, Water Hazzards, and environmental objects react using the same distortion system when applicable.

How the Extension Enhances Gameplay

- Brings a believable and reliable physics system to a 2D world with noise animations to make everything feel alive
- Allows for shared deformation logic between the player, enemies, and environment.
- Enables procedural animation instead of fixed sprite sheets which saves on memory and enables interesting dynamic visuals.

Target Audience and Platform

- **Target**: Fans of 2D platformers and the visual style of the slimes in slime rancher.
- **Demographic**: Teens to Young Adults (13-25)
- **Platform**: PC, specifically Windows and potentially Linux and Mac ports later on

Visual Style

- A vibrant, colorful, cartoonish pixel-art world (Keeping the design simple)
- Emphasis on soft bouncy motion and clear colors that make it easy to tell what's going on and sprites apart from each other.
- Smooth procedural deformation conveys elasticity and constant motion.

4. Technical Implementation Plan

Development Timeline

Week	Milestone
Week 1	Initial C++ project setup, GDExtension registration
Week 2	Mesh generation and update system Physics deformation and noise implementation
Week 3	Godot editor integration (parameters, texture) Game prototype integration (player slime)
Week 4	Environmental deformation and polish Final testing, optimization, documentation

Technology Stack

- **Language:** C++ for the extension source code and C# for in engine game logic scripts
- **Framework:** Godot 4.5 GDExtension API
- **Dependencies:** Godot C++ Bindings

Testing & Success Metrics

Unit Testing:

- Validate mesh deformation math
- Confirm editor parameter persistence
- Stress test with multiple deformable entities

Success Criteria:

- Stable frame rate (more than 60 FPS with 10+ deformed objects)
- No visual tearing or mesh corruption
- Consistent deformation sync with physics

Risk Assessment

Risk	Mitigation
Performance bottlenecks with many deformable objects	Limit subdivisions dynamically
Mesh corruption due to bad indexing	Implement vertex bounds checking
Desync between physics and deformation	Use fixed timestep updates
API version mismatch (Godot updates)	Keep GDExtension API isolated and modular

---

5. Team Collaboration Plan

Roles & Responsibilities

1. **Cole:** C++ extension core development, gameplay outlining, performance optimization
2. **Tyler:** Gameplay integration, testing, and collision event linking
3. **John Zeledon:** Visual polish, UI setup, and testing

Communication Protocols

- Weekly progress meetings via Discord
- Asynchronous updates through messages on Discord
- Code discussions via GitHub Issues

GitHub Workflow

- The main branch will be home to stable builds
- Each team member will have their own development branch for ongoing features they are building
- Dev Branch Naming Convention: {MemberName}DevBranch
- Pull requests must be reviewed by at least one team member before merging.

Conflict Resolution

- Resolve disagreements through structured discussion
- Majority agreement or instructor mediation if needed

---

6. References and Resources

Documentation & Tutorials

- Godot 4 GDExtension Docs: https://docs.godotengine.org
- Godot CPP Bindings GitHub: https://github.com/godotengine/godot-cpp

Inspiration

- Slime Rancher (Monomi Park)
- Gish (Cryptic Sea)
- Celeste (Maddy Makes Games)
- Godot Community Soft Body Simulation Prototypes