Simulation and Animation
Practical 3
Technique Documentation

Tank McShooty by [Unnamed Assignment Group]

Lukas Neuhold, 01430622
Lucchas Ribeiro Skreinig, 01431399

June 2020

1 Overview

The assignment was to create a computer game which featured at least 4 techniques listed in the introductory lecture. The following techniques are realized in the game:

1a The moving tanks follow an interpolated path at constant speeds.

1b The flowing particles are accelerated along a vector field path.

2 The tanks and movable objects exhibit linear and angular momentum changes according to rigid body dynamics.

3 The goal object has children, which rotate and orbit a point, exhibiting hierarchical transformations.

4 Destructible boxes are shattered according to Voronoi fracturing.

2 The "Engine"

Everything built upon the **GameMain** class was implemented by us. The system is a component based game object system. These game objects are hierarchically structured with transforms, building Scenes that have a root transform. All game objects, if not specified otherwise are parented to this root. Components implement certain behaviours and can implement different interfaces to, in a way, subscribe to the different update loops the engine provides.
There are multiple update loops a component can implement. The fixed update loop, implemented with the **IFixedUpdate** interface is the loop that is decoupled from the frame update loop. The fixed update rate dictates how often this loop is called. A update rate of 1 ms would mean that the fixed update

loop is called every millisecond. The update loop is the default Mono Game update loop, running at around 60 fps. There is also a late update loop, which is called later than the update loop, but still only once per frame. At the end of the update loop the draw visitor is dispatched to traverse the scene hierarchy, which ensures everything is drawn, that needs to be drawn.

3 The Code for the Techniques:

The implementation was done in C# with Visual Studio 2019, and a .sln file is included for simple loading of the project.

The code for **technique 1a** is contained in the class **PathFollower** (building the arc LUT) as well as the class Path, for the Catmull-Rom Spline calculation, and the class **VectorMath** for the implementation of the Catmull-Rom Spline. Path and Pathfollower are located **Content/Objects/Path/*.cs** The catmull rom implementation is in the file **Util/Math/VectorMath.cs** The arc LUT is built by sampling the curve at a certain sample rate and calculating the chord length of adjacent points. After this sampling the LUT is built by normalizing the chord lengths by the total length, and using them as keys, to map to the corresponding parameter values t. In the update loop, the path follower calculates the new distance he wants to be at and uses the LUT to figure out the best fitting t of the curve. With an exact match this is used, otherwise we linearly interpolate between the two closest options. **Path.cs** takes the t calculated by **PathFollower.cs** and uses it to figure out the Catmull-Rom Spline based position, which is then set as the path followers new position.

Technique 1b is realized in the **Content/Objects/Field/VectorField.cs** component, which takes the function Eval defined in **GameMain.cs** and evaluates the velocity of particles accordingly. Eval is defined as

$$Eval(X, Y, t) = (\cos(X + 2 * Y) * t, \sin(X - 2 * Y) * t)$$

The VectorField component searches for particles in the scene and calculates their velocities with the fourth-order Runge-Kutta technique according to their position and their lifetime, since particles live for 1 second. Since the position is defined relative to the cursor, the player can adapt the vector field dynamically.

Technique 2 is implemented in the BoxCollider component and the RigidBody2D component found in **Content/Objects/Player/PlayerComponents/*.cs**, as well as in the Velocity property of Transform.
The BoxCollider spans an axis-aligned bounding box around a GameObject's sprite and tests for collisions with other GameObjects. If colliding GameObjects contain a RigidBody component, and are kinematic, they enact forces on one another. Linear momentum calculation is based on the formula for elastic bounce and angular momentum is enacted on an object according to which side of the bounding box is collided with and the direction of the incoming force.

Newtonian gravitational pull, which can be seen as a component of the spiked ball can be found in **Content/Objects/Items/*.cs**

The **third technique** is contained in the function **CreateGoal** inside the class **GameMain.cs** as well as the function **AddGoalChildren** which recursively adds children to the previously created game object to a certain depth. Transform is located at **Core/Transform/Transform.cs** and is the class handling this hierarchical structure. Hierarchical transforms are the basis of our entire engine, and are supported to an arbitrary depth. For every position assignment, that does not directly assign the local position, we need to traverse this Transform hierarchy and transform the world position to the local space of an object. This comes with obvious performance draw backs when you have obscenely deep hierarchies, so it is recommended to set the local position directly. During update loops one should also not constantly query for the world position, but get the position once at the start of the loop and set it at the end, to minimize hierarchy traversal. A scene, found in **/Core/Scene/Scene.cs** in our game (even though rudimentary otherwise) defines a root Transform, where all other game objects are parented to, if not explicitly set to something else. This hierarchy is traversed multiple times by our engine for our different update and draw loops. These visitors can be found in **/Core/Scene/*.cs** and are dispatched in the update loop of **GameMain.cs**

Technique 4 is contained within the files:

- Content/Objects/DestructableBox/DestructableBox.cs
- Content/Objects/DestructableBox/Components/Destructable.cs
- Content/Objects/DestructableBox/Helper/Shatter.cs

As well as the folders:

- /Util/GeometryUtility
- /Util/Voronoi

Destructable.cs is the class that starts the shatter procedure avalanche. It checks if the box was hit, and on what side. From this information it generates voronoi seed points, and forwards this information (and other use-full information) to **Shatter.cs** This static class now takes the information gathered and engages the **ThreadedDataRequestor** (a helper class from one of Lukas' other projects adapted for this project) and request the polygons from the shattering with the given seed points from **Voronoi.cs**. **Voronoi.cs** does some pre-processing of the data, do make sense of the data it gets back from the library that builds a Voronoi Diagram with the Fortunes Algorithm. This is an MIT licensed library. With the data, which is an edge list of all the edges in the diagram, **Voronoi.cs** gets back from the library,it starts extracting the individual polygon points unsorted. After that it makes use of Lukas' geometry

utility library and the convex hull algorithm in that to build usable Polygon structures, from the unsorted point cloud that make up the Voronoi polygon.

This data is returned to **Shatter.cs** via the **ThreadedDataRequestors** call back. Which immediately dispatches another thread, that builds textures from those polygons (as we are using the default sprite batcher from mono game). These textures are only textured where the polygon contains the texture point, they are transparent otherwise. This is not an ideal solution, but workable as sprites are only 32x32 pixels in our game, and we only ever create 3 to 9 shattered pieces as well as doing all of that in separate threads. There was an git branch that actually had 2D polygons created from the returned data, but it stopped working and the bug was never found, and reversing the entire thing was not feasible anymore. So this is somewhat of a workaround, however efficient enough in the context of the project.

With these textures game objects are created and they are given a rigid body and some directional force to make the shattering more obvious.