Avalanche Simulation in a Particle System

As a part of the Master - Module 3D-Animation in the Hochschule Rhein Main purely written in Python and OpenGL

Tiras Zemicael
Karl-Blum-Allee 57, 65929 Frankfurt am Main
Tiras.Zemicael@student.hs-rm.de

Simon Rininsland
Roseggerstrasse 5, 65187 Wiesbaden
Simon.Rininsland@student.hs-rm.de

ABSTRACT

(What did we do. As tiny as possible)
- the question(s) you investigated (or purpose), (from Introduction)
- state the purpose very clearly in the first or second sentence.
- the experimental design and methods used, (from Methods)
- clearly express the basic design of the study.
- Name or briefly describe the basic methodology used without going into excessive detail-be sure to indicate the key techniques used.
- the major findings including key quantitative results, or trends (from Results)
- report those results which answer the questions you were asking
- identify trends, relative change or differences, etc.
- a brief summary of your interpetations and conclusions. (from Discussion)
- clearly state the implications of the answers your results gave you.

An Avalanche. A natural dreaded force of many snow and ice particles rushing down a Slope, driven by the Gravity. As many as snowflakes and ice particles which are included in an avalanche as good as we can play with them in an Particle System. One of the best examples for dynamicly rendered simulations for Particle Systems a snow Avalanche will be the central Part in our Project.
In order also to start just from the basics we decided to not use huge frameworks and start from the OpenGL Scatch. We will just use OpenGL Basics.

We will solve some Physically based Problems which comes around with the Topic of an Avalanche like:
- Particles with seperated masses, driven by a force.
- Physically Effects, bouncing Particles and combining ones.
and some OpenGL based Problems like:
- shadow for every seperated Particle
- performance Issues and optimization.

We have developed a piece of Software, we want to simulate a physic driven avalanche. The Core features are mainly dedicated to understand and solving physical Problems. We created Particles which reacts on physical forces from outside. These happens physically correct. After that we took some work to give the Particles a good-looking view, which should give a better understanding what we try to simulate at the first look.

1. INTRODUCTION

In the current time Simulation for almost everything exist. Simulations are a important part of todays society and is used for everything that canâĂŹt be test on a large scale or canâĂŹt even be create. The simulations are created to see possible situation or behavior of different thinks, for that the simulations need to be precise. Simulations are for the most parts not only a programming problem but also a problem in the subject the simulations are aiming for. Simulations need to behave the exact same as the real world counterpart, therefore a lot of physics is involved in such a simulation. And one of the highest requested kind of simulations are for weather phenomena. For that kind of problem simulations for every kind of weather phenomena exist. There are simulations for tidal waves, thunderstorms, flood and for our project Avalanche.

1.1 Project Description

1.2 Project Result

2. TECHNOLOGY

2.1 Python

2.2 OpenGL

2.3 Other Packages

3. ARCHITECTURE

Our Program inherits different Classes. The main Architecture can be described as follows:

All Objects in the World are described in the Objects class. Now, we have a Terrain and many particles. Both are implemented through the Object Class. A Particle inherits from the Objects Class, but uses some of the functions from the Objects Class. The World Class holds the Modell of our Program.
To have a Modell in our Program we need places to hold our Data. We took Numpy to store the position of every Particle in a global Grid-System.
Our Architecture must inherit a place to store all Particle Positions and must depict the Terrain. Because we saw fast, we can implement the Terrain Storage in a 2-dimensional Terrain, we decided to use two Arrays to store our Data.

3.1 Main Class

In every Program a Main Class is used to setup and initialised the Program. In our case, we use it to include all used external sources like OpenGL or our external OBJ-Loader, setting up the Scene, enable Options in OpenGL, registering mouse and keyboard functions and build the Window. In every loop Step from OpenGL we call the Objects Class draw function to enforce the calculation and displaying for the new Frame.

3.2 Object Class

The Object class holds many of the information dedicated to an object in our View. At the moment, we have two kinds of objects. The Terrain and our Particles. Bot need some properties and functions which are coded in the Object Class.
Some common Functions:

3.2.1 The Draw Function

The Object Class holds the draw function, which is the entry point for each loop Call from the Main Class. The draw function uses the draw function from the external OBJ Loader and does the translation for the position at each frame. Furthermore, it checks if the given Object is an Object (the Terrain) or a Particle. To only calculate the new position for the Particles and not for the Terrain.

3.2.2 The getHeightmap Function

To calculate the heightmap for the Terrain, which is stored in the World Class, we need to calculi it somewhere. We wrote a function to loop through all the faces in the obj file and map the y value of it on the x and z Index of the 2-dimensional Array with:

```
x = int(round(vertice[0]))
+ world.worldSize
y = vertice[1]
z = int(round(vertice[2]))
+ world.worldSize

world.terrainHeightMap[x][z] = y
```

Because of rounding errors in the height, we had some false Zeros in our Array which gets fixed afterwards with calculating the middle sum of the both neighbours from a face.

3.2.3 The getBound Function

@Tiras

3.2.4 The collisionDetection and collisionResponse Functions

These 2 functions are only abstract methods to get implemented by the particle Class.

3.3 Partcile Class

The Particle Class inherits from the Object Class and can use all the properties and functions from the Object Class.

3.4 World Class

Our World Class stores the main Data in our Program. We have used a direct mapping from the real view World to our Modell. The Terrain has a 128 Unit Width and Length. On the top, we need 2 Units to have an invisible Wall. Regarding these two Unit Rules, we need to have a 130 Units big 3-dimensional Array. Because Numpy is very fast in creating those huge arrays, we used Numpy to create the Array. With:

```
self.grid = np.zeros((self.gridResolution,
self.gridResolution, self.gridResolution),
 dtype=int)
```

We create fast a needed Array filled with zeros in Integer Type. We use Integer Types because we only need Integer values in our Array. Why is described later. Because we used -1 to set a Grid as not used, we must fill the array with:

```
self.grid.fill(-1)
```

with -1.
This creating and filling 3D Array with Numpy used less than a tenth of a second on a i7 PC. Creating this array and filling with -1 with conventional Python tools would spent more than 20 Seconds.
In nearly the same way we can create our terrain Map, with the difference of creating just a 2-dimensional Array with Numpy with Float Values in it, with:

```
self.terrainHeightMap = np.zeros((
self.gridResolution, self.gridResolution),
dtype=float)
```

Our World is now ready to hold all our needed Data for further calculations.

3.4.1 Grid System

We recognized very soon, we need a Grid system to not check the collision against all Particles. With more than 100 Particles the Program already gets unsmooth with less than 10 Frames per Second. The Idea was to only check Particle Collision between Particles which are in the same Vortex. A collision between 2 Particles which are in totally different Areas in the Map is not possible.
In the world - Class we already defined a grid for our world. Because the position of a single Particle is always written relative to zero and spawns between -64 to +64 we need to do a:

```
x = int(round(self.position[0]))
+ world.worldSize
y = int(round(self.position[1]))
+ world.worldSize
z = int(round(self.position[2]))
+ world.worldSize
```

for every Particle Position to map it on our Grid System. We use this x, y, z Indices for selecting the Vortex in our 3-dimensional Grid array. Furthermore, we check the value in the selected Vortex. If itâĂŹs -1 we can simply write the Index of the Particle. If there is already another Index from a Particle, we need to check if there is a Collision. How this works is described in another Topic. If there is a Collision, we need to Call the Collision Reaction from both or more Particles in the Grid. Afterwards we need to add our Index to the other Indices in the Grid. Of course, we must remove our Index from the Vortex before, after entering a new one. To have fast remove and append Array operations we use Numpy here too.

3.4.2 Terrain Height

Our Terrain Height Map works slightly different like the Grid System. Because the Terrain Map Array is depicting only the X and Z Axes, we only need to take the X and Z Value of each Particle and do the same like above:

```
x = int(round(self.position[0]))
+ world.worldSize
z = int(round(self.position[2]))
+ world.worldSize
```

We give a look in the Terrain Map with the Indices of x and z and check the value with the height of the Particle. We combined this âĂIJcould be a collisionâĂİ and recognize a real Collision in this case. The detailed check collision with Terrain is described later.

4. COLLISION

4.1 Particle Collision

4.2 Terrain Collision

5. CONCLUSION AND OUTLOOK