Creating a Foundation for a Snowboarding Video Game

Jared (Jake) Stookey

Abstract

In this paper, I describe a basic set of data structures and algorithms for creating a snowboarding video game. I use a simple model for the terrain and the snowboarder, and the physical laws that govern the interactions between them. For the terrain, I create a heightmap that is displayed. The heightmap can report interpolated height values and normal values at any point on the map. The snowboarder is defined by its mass, position, and velocity. Gravity and ground forces are applied to the snowboarder in such a way that results in a fast, realistic, and fun snowboarding simulation. The methods described here could be used in a wide range of video games that involve objects interacting with terrain including surfing games, marble games, motorcycle games, and many others.

1 Introduction

1.1 Motivation

A model for a snowboarding game should model the fun that is had while snowboarding. Exploration and freedom to roam anywhere on a mountain are what make snowboarding fun. Snowboarding games often restrict that freedom to predefined race courses. In this paper I explore ways to improve upon existing models for making snowboarding video games by giving the player more freedom.

1.2 Related Work

There are several snowboarding (and related) games available. I played the games listed here as an occasional reference while working on this project. One problem with each of the games is that a player does not have the freedom to explore the mountain. Instead, the limits of each course are strictly defined. This is one area that I would like to improve upon in my project.

- 1080 on the Nintendo 64
- Tux Racer for Linux
- SSX for Playstation 2

Fishman and Schachter (1980) developed a method for displaying heightmaps [4]. Coquillart and Gangnet (1984) discuss techniques for surface interpolation, clipping, and hidden-surface elimination of heightmaps [3]. Zhao and van de Panne developed a user interface for interactive control of snowboarding and other characters [2].

2 Data Structures

2.1 Heightmap

Heightmaps were used to represent the terrain in my simulation. The heightmap is a grayscale image stored in the Portable Network Graphics (PNG) format. White represents the highest possible point, and black represents the lowest possible point, and shades of gray represent height values in between. A heightmap can be displayed on the screen, and it can also be queried for height values and normals at any point within the height map. Even though the heightmap consists of a grid of Width x Height integer height values, the heightmap can be sampled at a greater resolution, and a floating point height value can be interpolated. This technique was used to create the illusion that the surface was smooth and continuous, even though the source PNG image is not smooth. As I worked on the project, there was an obvious need for interpolating the height value. Without interpolation, the snowboarder appears to be climbing stairs when the heightmap below him is displaying a much smoother slope. I had expected there to be a need to interpolate normals in a similar fashion, but I was pleasantly surprised to discover that discontinuous normals had much less of an impact on the smoothness of the snowboarder's motion. Therefore, I did not implement interpolation of the normals, although it would be an easy addition to make and should make for a more realistic simulation.

2.2 Object Model

I used the .obj file format to create the snowboarder. The .obj file importer would be useful for importing any 3d models into the game including trees, buildings, and other features. The .obj file format is very common and easy to understand. Most modeling tools are able to import and export the .obj format including Makehuman and Blender, two tools that I used to make the game. I made use of the .obj format's vertices, texture coordinates, and precomputed normals. The object can be loaded and displayed. The caller is responsible for orienting and sizing the model.

2.3 Snowboarder

The snowboarder primarily consists of a position, a velocity vector, and a mass. The snowboarder also maintains a history of previous positions, which is used in orienting the camera.

3 Algorithms

3.1 Heightmap Interpolation

To interpolate the height, I take the average of the 4 nearest PNG pixels weighted according to the snowboarder's relative proximity to each pixel. This is the area weighting interpolation scheme that was described in [1]. Figure 1 shows a picture of the height interpolation. P1 through p4 represent the pixels that make up the heightmap PNG. The point S represents the snowboarder's position in relation to the heightmap. The inner (lighter colored) square is the same size as an image pixel. It represents the total area. A1 through A4 represent the area of each pixel that is overlapped by the "total area" square. To find the interpolated height at S, I take the weighted average using the following formula:

- (A1*H1 + A2*H2 + A3*H3 + A4*H4)/A;

Where H[n] represents the height at P[n], and A represents the "total area".

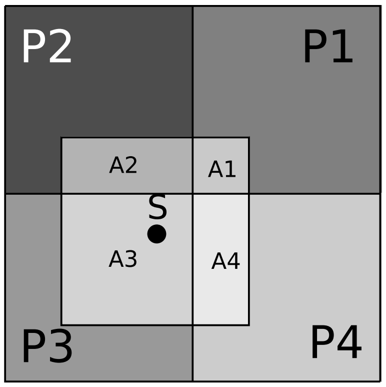

Figure 1: Calculating the interpolated height

3.2 Snowboarder Physics

I use a simple, but effective physical model to represent the snowboarder's physics. All of the snowboarder's movement is generated by the interactions between the snowboarder (a moving point), gravity, and the heightmap's surface.

Here is the key for the formulas in this section:

p	Position
v	Velocity
F	Net Force
wf	Weight Force
nf	Normal Force
a	Acceleration
M	Mass
t	Time Increment

Here is the algorithm for the snowboarder's physics:

1. Update the snowboarder's position
   - p = p + v
2. If the snowboarder is below-ground
   - Fix his height to match the heightmap
   - Absorb his velocity along the normal
   - Rotate the normal around the velocity if the user steers left or right (see the User Input section)
3. Calculate the Net Force (See Figure 2)
   - F = wf + nf
4. Calculate the Acceleration
   - a = F/M
5. Update the snowboarder's velocity
   - v = v + (a * t)

6. Rotate the model to point in velocity direction
7. Draw the snowboarder

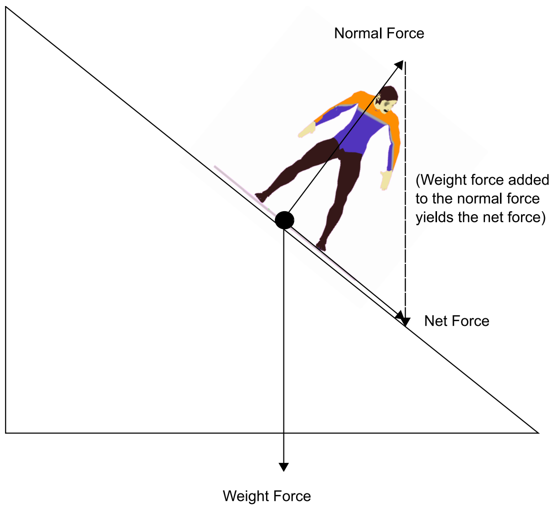
Normal Force
(Weight force added to the normal force yields the net force)
Net Force
Weight Force

Figure 2: Summing the forces

The most powerful part of the physics simulation occurs in the step, "Absorb his velocity along the normal". This was added to the code to fix a very small problem at a time when the physics were not working very well, and it turned out to fix most of the problems that my physics model was having. Absorbing the velocity along the normal causes snowboarder to deflect off walls and lose velocity when he hits a slope head-on, which emphasizes the effect that when he gradually touches down he maintains almost all of his speed.

3.3 User Input

The user can steer the snowboarder left or right. In order to approximate the complex physics involved in the way a snowboarder actually turns, I came up with a very basic model. If the snowboarder is touching the ground, then the snowboarder will accelerate in the downhill direction. That direction is based on the normal of the slope underneath her. During the act of carving, a snowboarder digs a side edge of the snowboard into the snow in the forward direction, effectively altering the normal of the slope below her. Using that reasoning, instead of using the normal of the face below the snowboarder to calculate the forces applied to her, I rotate that normal around her velocity, which rotates the downhill direction to the left or the right and causes her to smoothly turn in the appropriate direction.

3.4 Camera Orientation

In order to ensure that the camera will usually keep the snowboarder in view, I keep track of a series of the previous positions that the snowboarder has been, and the camera follows along behind the snowboarder, always pointing at the snowboarder. That way, I can ensure that the camera will never go through a wall, because the snowboarder can never go through a wall. This can be done without much calculation because I don't need to do any collision detection of ray tracing. There are still situations when the camera could be obstructed for a short while. Also, I actually position the camera behind the snowboarder's previous position in order to avoid putting the camera inside of the snowboarder's mesh, and that can result in the occasional situation where the camera is obstructed.

4 Results

In order to test my snowboarding game, I created an application using OpenGL for the 3D rendering, and Simple Direct Media Layer(SDL) for processing keyboard input and windowing. The project took a total of 120 hours.

My method for creating a simple snowboarding game resulted in a very satisfying simulation. The snowboarder will meander through the scene, following the path of least resistance, flying off jumps, deflecting off walls, and zigzagging bumpily through moguls. If he goes up a steep hill, he will slow down, turn around, and come back down the hill. The user can steer left and right, and the movement is generally very natural looking (see below for the special cases when it does not work properly). The code runs fast, so it leaves a lot of room for additional game content. The camera angle works very well, and creates a smooth animation that is easy to follow. Figure 3 shows a series of images taken while the program was running.

There are currently a few bugs, many of which should be easily fixed:

- There is no friction, so the snowboarder can accelerate to outrageous speeds.
- The snowboarder's turning ability does not work properly when he comes to a halt. The

snowboarder can spin in very fast circles. Instead, turning should be proportional to his velocity, so it should stop when his velocity is zero.

- The snowboarder does not like sharp creases in the terrain. Sometimes when one is encountered, his velocity changes direction suddenly.
- Time is not correct, so the simulation runs a different speeds on different machines.
- Vertical slopes cannot be represented directly in my heightmap, so I am unable to create terrain with a half-pipe that will allow the snowboarder go up a side, fly into the air, then return into the half-pipe again. Instead, the snowboarder will always fly out of the half-pipe.
- If the snowboarder is flying through the air, and the user is turns left or right when the snowboarder hits the ground, the normal is affected and the snowboarder can bounce in an unnatural angle.

5 Future Work

There is plenty to be done to make this simple game into a complete game. However, I will focus on the work that would refine what has been done in this project.

- Add friction.
- Add a shadow to give an indication of altitude when the snowboarder is in the air.
- Add skeleton physics to make the snowboarder respond to external influences.
- Add an adaptive time step so that the simulation runs at the same speed on all computers.
- Tilemap the heightmaps, so that a continuous varied terrain could be represented.
- Come up with a method for creating vertical slopes, so that a half-pipe will work properly.
- Come up with a way of piecing together the heightmaps in such a way that a large mountain could be modelled. Currently, the heightmaps lie on a single plane, so that only a single face of a mountain is represented. It would be much better if the heightmaps could be shaped into a mountain, allowing the user to explore all the mountain faces in a continuous model.

References

[1] Foster, N. and Metaxas, D., “Realistic Animation of Liquids”, Submitted to GMIP, 1995.

[2] Zhao, P. and van de Panne, M., “User interfaces for interactive control of physics-based 3D characters”, Symposium on Interactive 3D Graphics 2005.

[3] Coquillart, S. Gangnet, M., “Shaded Display of Digital Maps”, Computer Graphics and Applications, IEEE, 1984.

[4] Fishman, B. and Schachter, B. “Computer display of height fields”, Computers and Graphics 5 (1980), 53-60.

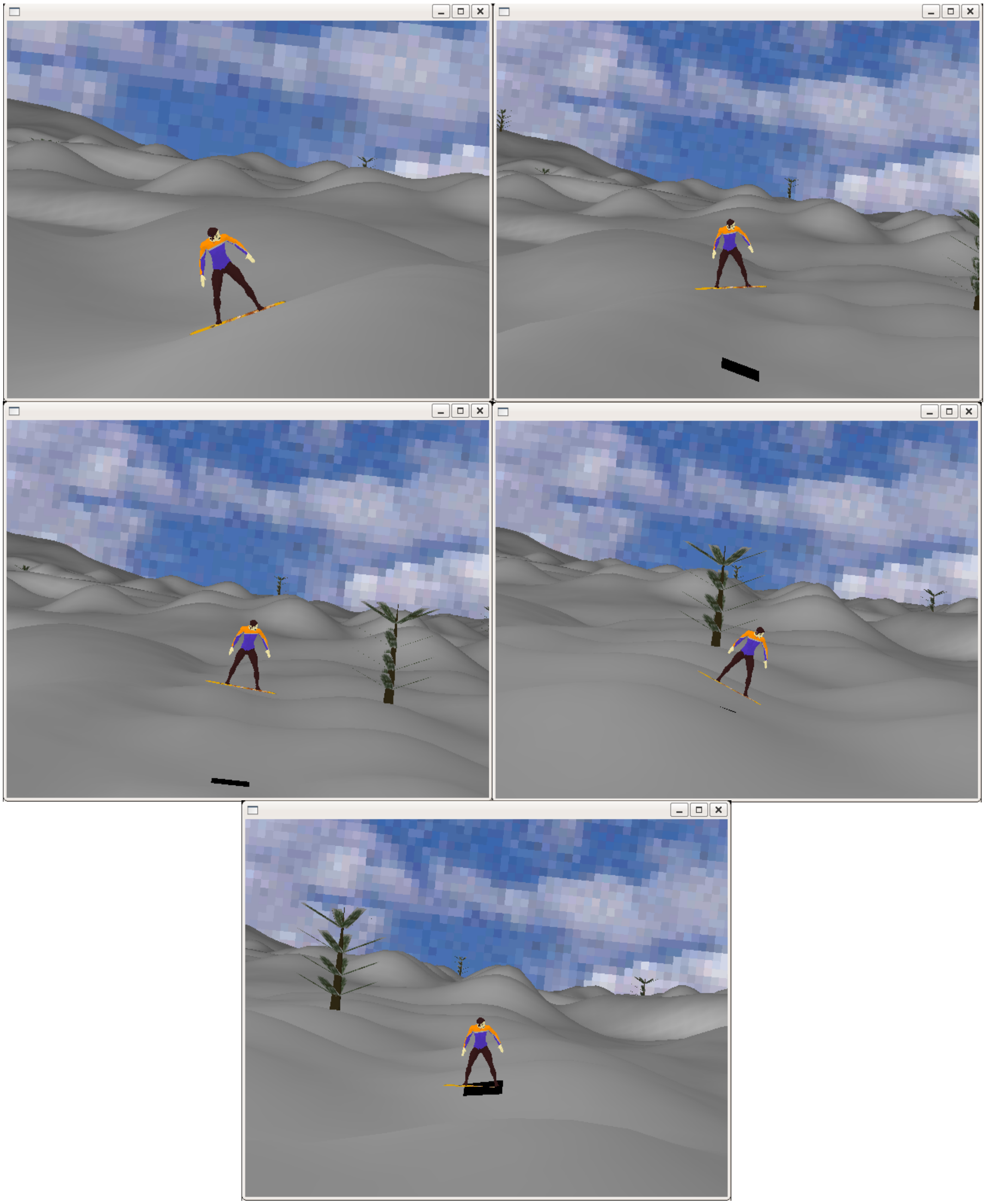

Figure 3: The program in action