Real-Time Physics Simulations on Portable Devices*

William Chargin[1] and Ulysses Sherman Bell[2]

[1]Department of Computer Science, Cal Poly San Luis Obispo
[2]Department of Computer Science, University of Texas at El Paso

15 August 2014

Abstract

The work of [3] uses the PhysBAM physics library and techniques in [5] to begin a real-time physics simulator for portable Android tablets. This application simulates a Jeep vehicle driving over flat terrain, jumping over a ramp, and pushing two spheres on the ground. We build on this application by developing new features for the simulation, rendering, user interaction, and underlying system code. Specifically, we add cloth and dynamic paint simulations, support for non-uniform terrains, more efficient collision detection, more realistic rendering, and high extensibility. Finally, we introduce a version of the application designed for more powerful desktop computers.

1 Introduction

In 2013, [3] began an Android application to demonstrate that PhysBAM, a C++ physics library, is capable of performing real-time simulations on low-powered, portable devices. We further develop this application to extend its capabilities.

We develop the physical simulations (section 2) by adding more simulation types, creating support for non-uniform terrains and multiple terrain types, and optimizing the existing simulations. New rendering features (section 3) include UV-mapped image textures and reflective materials, such as ice. The gameplay (section 4) is improved by allowing for more complex levels, including levels with non-physical interactions, and by introducing a precise control scheme. System improvements (section 5) mainly consist of making the code base easier to work with, through modularization, source control, and automation. We will discuss each of these aspects in detail in the sections that follow.

Additionally, we have created a PC version of the application that runs on desktop computers. It is easier and faster to develop for desktop computers than for tablets, so having a working PC application speeds up the development process and makes it easier to test new ideas. Eventually, all of the features on the PC application can be migrated to the Android application; most already have been.

Finally, we describe in section 7 how this research can be beneficial to the army.

2 Simulation features

2.1 Terrain

The terrain of the final product of [3] was simply a flat plane. We use two methods of terrain generation to allow for more complex and realistic custom terrains.

The first method is a height map. On a mathematical level, a height map is a function on $f : D \to \mathbb{R}$ that encodes the height of any point $(x, y) \in D$, where

*The authors wish to thank principal investigator Dr. Ron Fedkiw, and mentors Bo Zhu, Matthew Cong, and Saket Patkar. We also wish to thank Jingwei Huang, a student at Tsinghua University in Beijing who worked on the project with us.

$D \subseteq \mathbb{R}^2$ is the domain of the ground. Height maps are effective data structures for terrains in physics simulations because it is trivial to detect collisions; to detect whether a point (x, y, z) is colliding with the ground of a height map f, simply check whether $f(x, y) \leq z$.

To implement height maps, we restrict D to a finite square domain $[a, b] \times [a, b]$, and use a grayscale image file with false-color data to encode the height of each point. White and lighter shades of gray correspond to high elevation, while black and darker shades of gray correspond to low elevation. Therefore, the dimensions of the image determine the resolution of the height map (a larger image defines a denser—more detailed, but slower to simulate—height map). Furthermore, because image compression is an exhaustively studied field, the file sizes of the height map images are negligible (on the order of 10 kB). Finally, we can use a low-resolution height map for simulation and a high-resolution height map for rendering; this gives a much more realistic image with virtually no loss of simulation accuracy, and is easily accomplished by using two differently-sized images.

However, one limitation of height maps is that each point in D must be assigned exactly one height. This prohibits features such as overhangs, ridges, or holes in the mesh, because the relation f fails the proverbial "vertical line test." In cases where a height map cannot adequately represent the terrain, we instead use a triangulated mesh representing the topology of the terrain. These meshes can be created in any standard 3D modeling package. We use Blender, because it is free, open-source, and has a lightweight footprint.[1] Blender also ships with a customizable procedural landscape generator, which we use to generate random landscapes based on specified physical properties.

Once the meshes are created, they can be exported to the standard Wavefront Object (`.obj`) format. These can be converted to a triangulated mesh (`.tri`) using tools in the PhysBAM library. Next, an implicit level set file (`.phi`) is generated from the triangulated mesh; this allows the PhysBAM library to quickly compute intersections with other objects. After these files have been generated, they can be loaded onto the tablet for use in the simulation.

[1] Blender's file size is about 200 MB, uncompressed. Compare this with other systems such as Maya (about 5 GB) or Inventor (about 8 GB). This makes it feasible to use Blender on computers with lower memory and CPU requirements. However, despite its small size, Blender is still a feature-complete software package, providing an entire 3D pipeline.

2.2 New simulation types

We have introduced two completely new simulation types to the application, in addition to the articulated rigid body systems previously in place. The first is a cloth simulation, and the second is a variant of a dynamic paint simulation. See fig. 1 on the following page for examples of each.

Cloth and soft-body simulations are extremely important for a realistic physics environment. From curtains and clothes to towels and napkins, cloth-like objects permeate our surroundings. We have chosen to introduce the flag of the US Army as our simulated cloth; within the game, reaching the flag is the goal. To implement a cloth simulation, which normally requires expensive finite element analysis techniques, we borrow techniques from mainstream game development, such as cached pre-bakes, billboarding, and alpha compositing. Under these optimizations, we have included a flag blowing in simulated wind that looks realistic while retaining a high simulation speed.

We have also added the ability for the vehicle to leave tracks in the ground where it has driven. This uses a simulation known as dynamic painting: imagine that the ground is a giant canvas, and each wheel is a paintbrush, so that the wheels leave marks wherever they contact the ground. However, our dynamic paint implementation uses displacement instead of coloring; that is, the vertices of the ground are moved down, not recolored. In the default sand terrain, this provides a more realistic look and feel to the simulation. Again, this exploits the difference between the simulation and rendering meshes: because we only displace the rendered ground, we don't have to recreate the level sets (which is infeasible in real time), nor do we have to modify any of the simulation parameters.

To implement the dynamic paint simulation, we need to determine the points of contact of the wheels with the ground. On a height map, this is easily calculated by taking the centroid of the wheels and

(a) The flag cloth simulation.

(b) Tracks in the sand.

Figure 1: Screenshots of new simulation types.

rounding it to the nearest lattice point on the height map image. This is an $\mathcal{O}(1)$ operation. On a triangulated mesh, we can achieve a relatively fast lookup time by using an octree data structure. This can be visualized as essentially "zooming in" on the correct octant of the terrain until only the closest vertex remains. In the expected case, this operation is $\mathcal{O}(\lg n)$ in the number of elements in the triangulated mesh.

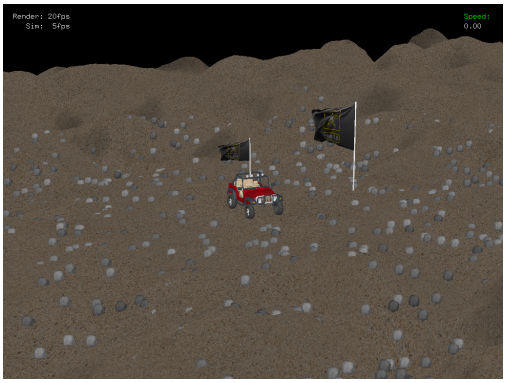
2.3 Hidden land mines

To add an element of danger to our application, we added land mines to our simulation. If the vehicle hits a land mine, it is thrown at a random velocity, and the vehicle is destroyed: the wheels fall off and the vehicle can no longer operate. Additionally, we added small gray rocks to the scene, identical in appearance to the land mines. This forces the user to make choices: act conservatively, avoiding all rocks, or risk a chance of hitting a mine. See fig. 2 on the next page for images from the game.

When adding rocks and mines, we distribute the elements uniformly over the terrain. With a height map, this is easy, because we can pick a random point $(x, y) \in D$, and find the point $(x, y, f(x, y))$ on the terrain where the land mine should be anchored. However, this is not feasible on a triangulated mesh, because there is no one-to-one mapping from ground points to surface points. Additionally, it is sometimes desirable to limit the distribution of land mines, for example, only on a road. This imposes an additional constraint on the simulation. To solve this, we adapt the Metropolis–Hastings algorithm for distribution sampling combined with a UV-unwrapped density map as a two-dimensional image (see section 3.1 for more information).

Consider the surface $S \subseteq \mathbb{R}^3$ of a triangulated mesh with n faces. Let each $\bar{f}_i \in S$ be the centroid of the face on the mesh with index i, for $1 \leq i \leq n$, and let $t : S \to [0, 1]^2$ be the UV mapping function. Then, if $\rho : [0, 1]^2 \to [0, 1]$ is the density function, we can sample k points according to algorithm 1 on the following page. By modifying line 5, this can be extended to linearly interpolate points on the faces. In cases where the average value $\iint_{[0,1]^2} \rho \, dA$ is small and the measure $\lambda(\{t(x, y, z) \mid \rho(t(x, y, z)) = 0\})$ is close to 1—that is, when most points are not on the density map—the triangulated mesh can be preprocessed, or the distribution from which random points are selected can be changed to a non-uniform distribution, to avoid excessive rejections.

Note that this assumes that the areas of the triangles with non-negligible density values have a low variance. This is true in our case, because Blender's mesh triangulation decimator can perform Laplacian smoothing. If this were not the case, the density function could be weighted by the ratio of the face area to the average face area. Alternatively, the mesh could



(a) The vehicle in a mine field. . . can you tell which rocks are mines?

(b) Mines highlighted for debugging purposes (in red).

(c) The land mine explosion and resulting chaos.

(d) The three main stages of the explosion.

Figure 2: Land mine setup and integration.

Algorithm 1 Point sampling algorithm, adapted from the Metropolis–Hastings algorithm.

function SAMPLE-POINTS($\bar{f}_i, t, \rho, k$) ▷ see section 2.3 for parameter descriptions
  $P \leftarrow \{\}$
  while $\|P\| < k$ **do**
    repeat
      $(x, y, z) \in \bar{f} \leftarrow$ RANDOM ▷ $\bar{f}_i$ for some random $i \in [1, n]$
      $(u, v) \leftarrow t(x, y, z)$
      $\alpha \leftarrow \rho(u, v)$
    until $\alpha >$ RANDOM ▷ random values should be in the interval $[0, 1)$
    $P \leftarrow P \cup (x, y, z)$
  return P

simply be re-tessellated offline with a smooth octree, and the old UV coordinates could be shrinkwrapped to the new model.

We also display a simulated fireball as a result of the explosion (see figs. 2c and 2d). This is simply rendered as a billboard facing the camera; otherwise, the image would need to be rendered as 3D voxel data, using gigabytes of memory and rendering far too slowly.

2.4 Simulation optimizations

To retain a high speed of simulation while adding more features, we needed to optimize the simulation code. One important factor in this optimization was the creation of a collision manager.

In the general case, a simulation with n rigid bodies has $\frac{1}{2}n(n+1)$, which is $\Theta(n^2)$ pairs of objects. A naïve simulator will need to calculate the collisions of all pairs of these objects, which will be very slow for arbitrary meshes. With our collision manager, we can explicitly specify which pairs of objects require collision detection. This prevent wasted intersection calculations, and provides a great increase to the simulation. To demonstrate this capability, we include two specific levels in our program. The first has a large number of rocks rolling down the side of the mountain. In this level, collisions only need to be detected on the rock–mountain and rock–jeep pairs; note that this is $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$. The other level is the land mine level described in section 2.3; this also requires only $\mathcal{O}(n)$ pairs. See section 6.2 for more information on level design.

Currently, this is implemented by manually selecting the pairs bodies on which collisions should be computed. In future work, this could be improved by constructing and mutating a persistent bounding volume hierarchy (BVH). Collision detection in a BVH is extremely fast because it is so simple, so this could be used as a preliminary test before more accurate collisions are computed. Furthermore, we could exclude pairs of objects where both objects are static. These two optimizations combined should provide a similar speed-up factor with a more automated approach.

Another important optimization, briefly mentioned in sections 2.1 and 2.2, is the separation of simulation and rendering objects. Throughout our application, many aspects of the program are simulated differently than they are rendered. For example, we use a low-resolution (64×64 pixel) height map for the simulation, but render a higher resolution (128×128 pixel) image. Because the height maps are generally relatively smooth, the simulation is still an accurate approximation. Similarly, we use convex approximations ("proxy geometry") for the more complex simulation models in the rigid body simulations, then bind the high-resolution models to the results of the low-resolution simulation. Essentially, these optimizations improve the speed of the simulation while also improving the visual quality of the result.

3 Rendering features

3.1 Image textures and UV maps

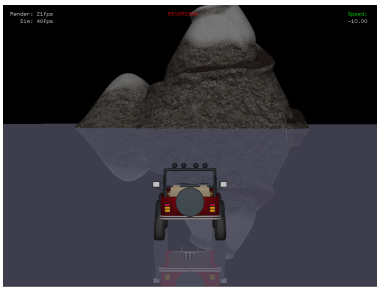
The use of non-uniform terrains, as discussed in section 2.1, makes the simulation more realistic. However, even a non-uniform terrain does not look realistic if it is of uniform color. Adding image textures to our environment adds greatly to the realism of the simulation with very minimal computational cost.

The tricky part of applying image textures is defining which parts of the image go where on the mesh. This is similar to the problem of trying to take a globe and flatten it into a map: you can't do it without cutting and stretching the surface. The computer graphics industry standard solution to this problem uses a technique called UV mapping, which defines exactly how the cuts (seams) and deformations should be constructed. Modelers carefully create UV maps to minimize and hide the seams while also minimizing the distortion.

In UV mapping, each vertex $p_i(x_i, y_i, z_i)$ of the mesh is assigned a texture coordinate value $t_i(u_i, v_i)$ in 2-space. Usually, we limit $(u_i, v_i) \in [0, 1] \times [0, 1]$, and apply edge wrapping when necessary. To draw the image on the mesh, we look at each vertex p_i on the mesh and apply the color at pixel t_i on the image.[2] We

[2]Of course, texture images are larger than 1×1 pixel, so we multiply by the dimensions of the image: the mapping of a vertex p_i for an image of width w and height h is (wu_i, hv_i).

(a) A UV-mapped sand texture on a mountain.



(b) A different texture on the same mountain as fig. 3a, with ice reflection.

(c) An example of the fog effect.

Figure 3: Examples of rendering improvements.

can then interpolate between points to get the correct face colors (either linear or bicubic interpolation is typically used, depending on the desired quality). See figs. 3a and 3b for examples.

Blender includes a system to allow modelers to define UV coordinates for their meshes. After this is completed, we pass the output through a series of scripts to generate a compact binary representation of the UV data (usually a few dozen kilobytes in size). This data can be loaded into our application and passed to the OpenGL renderer to correctly display the image textures.

3.2 Reflective materials

After creating support for multiple terrain types, we decided to add ice as our secondary terrain, because it contrasts with sand in many of its physical qualities. Of course, ice has a high albedo, around 0.6 for bare ice and 0.9 for snow-covered ice.[3] Therefore, we designed and implemented an efficient material reflection scheme. Our scheme employs a clipping mask to limit the canvas to the ice, and then mirrors, translates, and renders the rest of the scene at an opacity value of $\alpha < 1$. The value of α is effectively the albedo. This does incur a performance hit in the render speed, but even the reduced render speed is much faster than the simulation speed, so the overall application is not slowed down.

See fig. 3b for an example.

[3]We approximate the mean value of the bidirectional reflectance distribution function (BRDF) as the albedo. While this is not globally physically accurate, it is a common approximation that holds locally under certain assumptions (for example, that the ice does not exhibit subsurface scattering, and that it is of uniform thickness).

3.3 Ambient effects

We further improved the game environment by adding fog rendering. Consistent with standard models, our fog causes the visibility level to fall off as $e^{-\rho d^2}$, where ρ is a constant representing the density of the fog. Currently, the fog only serves as an aesthetic effect. However, with further development, the fog could be the basis for an accurate weather/environment simulation. (See section 6.2 for more information on level design.) See fig. 3c for an example of fog in the game; note how the terrain fades out as it gets farther from the camera.

4 Gameplay features

4.1 Interaction with the environment

The vehicle–terrain interactions are clearly part of the physics simulation itself. However, we have created other interactions as well. Often, the user is required to perform one of these actions to complete a level.

For example, one of the levels is intended to teach the user how to drive forward and backward in a

controlled manner. The user must drive through a narrow corridor, and push a spherical ball off a cliff into a funnel. The ball rolls down the funnel, where it hits a button. Once the button is depressed, a gate at the start of the level is allowed to open, and the user must drive backward through the gate to get to the finish line.

In this level, we have a number of physical interactions: vehicle–ground, ball–ground, ball–funnel, and funnel–button. These object pairs are explicitly programmed into the collision manager (see section 2.4). Additionally, we have a non-physical interaction that affects the physical world: when the ball hits the funnel, the ϕ-rotation constraint preventing the rotation of the gate hinge is removed. This demonstrates that our software is capable of handling non-trivial physical and non-physical interactions while retaining the real-time speed of the simulation.

4.2 Control scheme

The original control scheme of the Android program was completely tilt-controlled (pitch for throttle, roll for steering), and the "gas" and "brake" buttons on the HUD had no effect. We implemented a "manual drive mode" that uses the gas and brake buttons for throttle and keeps roll for steering. This provides a more intuitive interface—the tablet itself is like a steering wheel, and the gas and brake buttons are at the bottom, just as the pedals in a physical vehicle—and offers more precision. A more precise control scheme makes it easier to control the vehicle, which makes the simulation more realistic, and also makes testing significantly easier. This control scheme is provided as another option; the original control scheme is kept as well.

5 System improvements

In addition to developing the application, we also improved the underlying system to aid future developers.

5.1 Modularization

The original code base mixed code designed for this specific application with code from the PhysBAM library itself. The PhysBAM library had been heavily modified to support this application specifically. This makes it hard to understand and debug the code; it would be akin to rewriting parts of the Windows operating system in order to create a single application.

To fix this, we removed code specific to our application from the PhysBAM library. In cases where the PhysBAM library could not support our needs, we implemented the required features generically, such that they could work with other applications or even be merged upstream into the main library. Some of our developments, such as height fields and support for UV textures, could be useful in other PhysBAM projects, and are portable in their current state. We extracted "magic numbers" (seemingly random constants appearing many places in a code base without any explanation) into more centralized locations, and separated blocks of code that didn't relate to each other. The result is a more workable code base that future developers should more easily understand.

5.2 Source control

Most large projects are set up with source control, which allows easy tracking, revising, and inspection of code by multiple developers on one team. One of the first things we did when we began this project was to initialize a Git repository and GitHub remote host. The benefits of using source control, and GitHub specifically, are vast. Multiple developers can modify different files on different computers—or even different parts of the same file—and have their changes automatically merged. Files are always visible online, in any of their previous revisions, so if something breaks, you can grab an old file or even an old version of the entire project. You can see who wrote or modified any particular line of code, making it easier to share information about how the code works. Tracking down bugs becomes much easier, because it is trivial to step through versions of the code looking for the last working version, and it is also trivial to analyze the differences between versions of the code. Finally,

issues, bugs, to-do lists, new features, and documentation are automatically tracked and managed through GitHub's commit, issue, and pull request systems.

5.3 Tools and utilities

To get the various aspects of our application to combine properly, we need to generate a lot of different types of files. We need to convert from `.blend` files to `.obj` files, and then we need to extract the texture coordinates into both ASCII and binary `.uv` files and strip the coordinates from the object files. We need to generate implicit level sets as `.phi` files, and sometimes rigid-body cache data as `.rgd` files. Finally, we need to transfer all of these files to the tablet so that they are available to the application.

This is clearly a long process, and it's exactly the sort of thing that computers are good at. Therefore, we developed scripts and utilities to automatically convert between these file formats and transfer them to the required devices. These scripts are available in a unified location from our development computers, and are also hosted on the aforementioned GitHub repositories. The inclusion of these tools has saved us hours of wasted time and frustration from simple human error.

6 Results

6.1 Performance

On the PC version of the application, the simulation speed hovers around 35–40 frames per second, while the rendering speed fluctuates anywhere from 40–60 frames per second, even while double-rendering the reflections as described in section 3.2. When rendering the explosions discussed in section 2.3, the rendering speed drops by 5–10 FPS, but the simulation speed is barely affected.

Note that the simulation and rendering FPS readings in the screenshots in this document are often inaccurate. The act of taking a screenshot slows down the rendering speed, and running in debug mode or on a remote host can cripple both readouts.

6.2 Level overview

In this section, we summarize the various levels currently implemented, to give a general idea of the possibilities offered by our program.

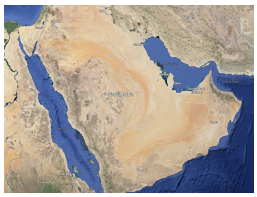
One level features an icy mountain, similar to the mountain in fig. 3b. The goal of this level is to drive up a path that winds up the side of the mountain. This is intrinsically somewhat difficult because the ice is slippery, and it's not easy to maneuver the vehicle. To add to the difficulty, we include a constant avalanche of boulders falling from the top of the mountain. This level demonstrates the ability described in section 2.4, simulating a large number of objects (dozens of rocks) at one time. It also demonstrates that it is possible to add and remove rigid bodies at any time, because rocks that fall off the bottom of the screen must be destroyed and moved to the top.

In another level, the user must cross a gap to get to the flag. The path to the left is a suspended rope bridge, which is unstable and narrow. To the right is a field of rocks, and by driving over this field, the user risks hitting a land mine. On a physics simulation level, the rope bridge is an articulated rigid body, comprising multiple planks adjoined by springs.

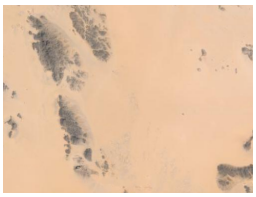
7 Conclusion

Real-time physics simulations on portable, low-powered devices have a wide variety of potential uses for the army. One potential use of our program, in its current state, would be simulated operation planning and practice.

Prior to conducting an operation, officers in the field could receive a virtual model of their environment on their portable devices. The virtual environment could incorporate GIS data such as elevation topology, foliage, and the material composition of the ground (dirt, sand, rock, etc.). This information is widely available, and could be incorporated (even automatically) into the existing tablet application by applying height maps, setting image textures, modifying coefficients of friction, etc. This would enable officers and soldiers to simulate the movement of vehicles through an unfamiliar landscape beforehand, which



(a) A satellite view of a region in Saudi Arabia. Image from Google Earth.



(b) The satellite image zoomed, cropped, and processed.

(c) A height map created by color-adjusting the image from fig. 4b.

(d) The in-game level, with terrain and image, generated by combining figs. 4b and 4c.

Figure 4: The process of converting a satellite image into an in-game level with height map.

would in turn allow urgent missions to be completed more quickly and more effectively. Figure 4 demonstrates the process of converting satellite data into a level compatible with our program.

With further development and more powerful portable tablets, our framework could be used to simulate an underbody blast. This is not currently possible in real-time, simply because a physically accurate simulation encapsulates too many variables to be feasibly solvable by today's portable, low-powered devices. However, the power and efficiency of devices is constantly increasing (Moore's law famously states, roughly, that efficiency doubles every two years), so this should be possible on the higher-powered devices of the future.

8 Future work

Possible improvements to the application include: false-color image maps bound to physical or material properties of the terrain, such as friction maps, specularity maps, and so on; a "multi-player" mode where multiple vehicles are simulated and controlled at once; weather, especially particle weather such as precipitation, that could impede vision; more types of vehicles, such as jets, ships, or amphibious tanks; optimized support for dense urban environments; a network link with desktop computers or clusters to perform more complex real-time simulations.

References

[1] Kevin Hawkins and Dave Astle. *OpenGL Game Programming*. Ed. by André LaMothe. Prima Tech's Game Development. Prima Tech, 2001.

[2] Mark Kilgard. "Rendering fast reflections with OpenGL". In: *OpenGL Code Resources*. Silicon Graphics, Inc., 2012.

[3] G. Jake Manning and Garrett Shaw. "Real-Time Physics Simulations on Android Devices". In: *Summer Institute Final Report*. Vol. 5. Army High Performance Computing Research Center, 2013, pp. 29–31.

[4] Andrew Selle et al. "Robust High-Resolution Cloth Using Parallelism, History-Based Collisions and Accurate Friction". In: *IEEE Transactions on Visualization and Computer Graphics* 15.2 (2009), pp. 339–350.

[5] Rachel Weinstein, Joseph Teran, and Ron Fedkiw. "Dynamic Simulation of Articulated Rigid Bodies with Contact and Collision". In: *IEEE Transactions on Visualization and Computer Graphics* 12.3 (2006), pp. 365–374.

[6] Rachel Weinstein, Joseph Teran, and Ron Fedkiw. "Pre-stabilization for Rigid Body Articulation with Contact and Collision". In: *ACM SIGGRAPH 2005 Sketches*. 2005, p. 79.