LEVI SCHUCK
STEPHEN ROLLINS
JEREMY WAINWRIGHT

REPORT ON

PHYSICS SIMULATION FOR ANIMATION

SUBMITTED TO
CARYN LESUMA
FOR ENGLISH 316

BRIGHAM YOUNG UNIVERSITY
PROVO, UTAH
APRIL 13, 2013

Contents

List of Figures

The Physics Animation Team

April 13, 2013

Brigham Young University
Rm 3004 JKB
Provo, UT 84602

Dear Master Lesuma,

We have written this technical report about physics engines to determine which engine best exemplifies the qualities that make an engine well-suited for movie animation. We have conducted our research from a combination of related articles, the enginesâĂŹ documentations, and our own original experimentation, and are now submitting it to you for grading. We are not planning on submitting this paper to any journals for publishing. This report is meant to be viewed by anyone who is interested in animation or physics engines, regardless of their experience in the field of animation.

As discussed in our paper, we compare three different physics engines, judging each primarily on its visual correctness in the experiments we perform. As a result, it is very easy for us to identify the strengths of each engine and what makes them more or less suited for film-quality animation. We discuss several aspects of physics engines, covering the topics of collision detection, constraints, discrete and continuous motion, and stability and errors. For conclusive results, please refer to the report.

Future research may be conducted on a wider variety of physics engines. We have learned much from our research and hope that you will benefit from it as well.

If you have any questions, feel free to contact Jeremy Wainwright at drumgeek13@gmail.com.

Sincerely,

Levi Schuck

Stephen Rollins

Jeremy Wainwright

Abstract

Animation is a field that demands high quality works. Animations must look accurate if they are to attract viewers into the world the animator is creating. A physics engine is one of the easiest ways to make the characters and objects in an animation move more realistically. There are a lot of physics engines available for use, so it can be hard to decide which one best suits the purpose of the animation. As such a key part of the animation process, choosing the right physics engine is one of the most critical decisions in planning any animation.

In this paper, we present three different engines, comparing them based on several qualities that make an engine fit for use in movie animation. These qualities include how they handle collisions, constraints, and settling. The three physics engines–Bullet, PhysX, and ODE–are all third-party physics engines used by big-name companies and individual developers alike to create realistic animations, both for producing games and films.

To gather empirical data for our study, we perform simple but practical tests that will check the performance of the engines in various high-stress situations. Based on the results of our tests, we conclude that Bullet is the engine best suited for film animation because of its accuracy and stability. PhysX is better suited for video game or cartoony physics, especially if speed is a critical factor. We then recommend further research that could look at a wider variety of physics engines or even focus on soft body physics like hair or cloth.

Introduction

When was the last time you went to see an animated film? If you've been to see one within the past 20 years, chances are that it was computer animated. Recall, if you can, a moment in one of these movies when something fell to the ground or crashed or collided with another object. Did it look realistic?

Creating realistic animations is a key goal in movies and games. There are several ways to achieve this realistic look. One way is to animate things by hand, **frame** by frame, using a reference object to make sure it looks authentic, see Figure 1.1. This often takes huge amounts of time and resources that could have been used to further improve the film or game in other ways. The first animated films were all hand-drawn, but now, with the help of computers, there is a faster way: using a **physics engine**.

A physics engine allows the animator to give the models certain qualities and attributes that tell the computer how to handle the model in a physically-based world. The engine calculates where each object is, whether or not objects are hitting each other, and how the objects should react when they do collide. Having the computer run the **simulation** saves time that would have been spent animating the movements by hand. Saved time means the animators can be more productive and the film can be finished at less cost. One of the greatest advantages of using a physics engine is **collision detection**, a key process to creating a realistic animation.

Computers have no innate concept for physical properties like movement, inertia, or mass. However, with programs like physics engines, it is possible for computers to emulate these physical properties. Physics engines are very popular in animation because they make the people and objects in video games and movies behave as they would if they were actually real. However, no engine creates a perfect imitation of reality. For example, most physics engines only simulate whole objects, taking no account for atoms or other basic building blocks in real life. The calculations required to create a per-

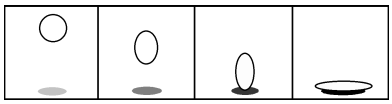

Figure 1.1: The movement of an animated ball, as drawn by hand, separated into frames.

definition 1.1 **Frame:**
A single snapshot, like a photo of the objects in motion. Several frames viewed in one second create the illusion of motion, much like a flipbook. A single frame represents the snapshot of all the motion that happened in a very short period of time.

definition 1.2 **Physics Engine:**
A computer program that uses well-known physics equations to calculate the positions and motions of several objects in a simulated world. Physics engines allow users to create an imitation of a real-life scenario to predict how real objects would behave.

definition 1.3 **Simulation:**
Giving the computer a scenario and telling the computer to make the objects behave as they would under the laws of physics. Running a simulation is like hitting "play" on a movie.

definition 1.4 **Collision Detection:**
Determining whether two objects are touching and calculating their reactions.

fect of a replica of real life would take longer than simply animating the scene by hand. Thomas Yeh et al. state that the time cost grows substantially with each object, collision, and calculation. [7] This means it is too hard to make completely accurate calculations, as they're too complex to be solved quickly.

The key to a fast but accurate simulation is to make the calculations efficient with an acceptable amount of realism. Thomas Yeh continues to say that "the results...do not need to be absolutely accurate, but do need to appear correct (i.e believable) to human users." [7]

Several physics engines are available for the general public to use for free, each taking a slightly different approach to the various processes involved in physics simulation. While no one engine is perfect for all situations, many are ideal for specific settings or uses. In this paper, we will be specifically considering how acceptable certain widely-used physics engines are for creating realistic animations.

WE HAVE CHOSEN THREE DIFFERENT PHYSICS ENGINES (Bullet, PhysX, and ODE) to analyze and compare what important qualities the engines have that make them fit for use in the animation field. Each of these engines have been built by different groups, and each was written with different objectives in mind. Bullet is an **open-source** engine that has been used in titles such as *Toy Story 3*, *MegaMind*, *How to Train Your Dragon*, and *Shrek 4*. [3] PhysX is a **closed-source** physics engine produced by nVidia, a well-known producer of **graphics cards** and related video technology. PhysX has been used in a long list of games, including *Age of Empires III*, *Gears of War*, the *Mass Effect* series, and the *Tom Clancy* series. [4] ODE (Open Dynamics Engine) is open source and is by far the oldest of the three engines we will be looking at. It was originally created by Russell Smith, a developer who now works at Google. It has been used in titles like *World of Goo* and *Mario Strikers Charged*. [5]

DEFINITION 1.5 **Open Source:**
Programs whose code is accessible to the general public. Typically, open-source programs are produced as a community effort rather than by a specific company, and–unlike most closed-source programs–are available for free. Mozilla Firefox is an example of an open-source program.

DEFINITION 1.6 **Closed Source:**
Programs whose code is not accessible to the general public. Closed-source programs are written by companies or small groups and are often used to make a profit. Often, closed-source programs have a more "professional" appearance and extra features when compared to their open-source counterparts. Google Chrome is an example of a closed-source program that is available for free.

DEFINITION 1.7 **Graphics Card:**
An essential piece of hardware that performs many of the intense calculations required to determine what is drawn to the screen and where it will be drawn.

IN THIS PAPER, WE WILL PRESENT a series of experiments intended to identify how each engine performs in five key areas: change of state, collision detection, calculating motion, stability-affecting shortcuts, and constraints. Each of these key qualities will be discussed in detail in the next section. To better understand these five qualities, we will present the results of the following experiments:

- A stack of rubble
- Cannonball shot at a wall
- Ball rolling down a set of stairs

Physics Engine Qualities

Changing States of Objects

Research into the improvement of animation physics has been a long-time pursuit of computer scientists and animators. Studies like [1] focus on the automation of the physics simulation process. A major goal in programming is to get the computer to do as much of the work as possible without needing to be assisted by the developers. In [1][1], they identified that "A considerable portion of the difficulty [of physics-based animation] results from the need to track and manage instantaneous changes in the **states** of objects and the equations and constraints that govern their behavior" Objects in a physics simulation are not self-aware. They do not keep track of their own motion. Like a mindless robot, the objects simply go wherever the physics engine tells them to go.

[1] Thomas Ellman. "Specification and synthesis of hybrid automata for physics-based animation". English. In: *Automated Software Engineering* 13 (3 2006), pp. 395–418. ISSN: 0928-8910. DOI: 10.1007/s10851-006-8532-4. URL: http://dx.doi.org/10.1007/s10851-006-8532-4.

DEFINITION 2.1 **State:**
The present condition of an object which the physics engine uses to decide how the object will behave, especially with gravity or friction. States often describe the motion of an object: motionless, rolling, sliding, falling, no gravity.

Because the engine is managing such a huge variety and number of objects, it must have an effective way to classify how each object is currently behaving so it can properly simulate the motion of each object. As stated in the above quote, there are two main factors of motion that the physics engine needs to be particularly concerned with: instantaneous changes and changes in constraints. An example of an instantaneous change would be a ball bouncing off of the ground. At the moment of collision, the vertical motion of the ball needs to be reversed, with some of the energy being lost as sound and friction (or else the ball would bounce indefinitely). Using this same example, a change in **constraint** would occur when the ball finally loses enough energy so that it no longer bounces. Instead of continuing to calculate the ball's new vertical velocity, the ball is constrained to remain in contact with the ground and roll until its forward motion is exhausted or it falls off of an edge (at which time its vertical motion constraint is lifted). In our study, the role of managing these equations and constraints is left up to the engines. A good engine will properly handle changes of constraints and instantaneous changes that result from collisions.

DEFINITION 2.2 **Constraint:**
Restrictions imposed on an object by preventing motion along certain axes, as with a ball only allowed to roll instead of bounce or a wheel only allowed to spin in one direction on an axle.

Efficiently Detecting Collisions

We looked at other studies which focused more on how to make the computer pickier about the calculations it does. In [6][2], the authors talk about how to do a process known as Collision Detection more efficiently. Collision detection is a vital step in physics simulation when the engine checks all of the objects to see if any of them have run into each other. What happens after a collision is a separate matter; collision detection primarily concerns itself with whether the collision has occurred or not. In this way, the process of collision detection is like an apprentice working for a master craftsman; he has an important job to do, but his work alone does not produce a masterpiece on its own. The process of collision detection must happen very quickly, and often occurs several dozen times per second. Collision detection becomes a more time-consuming process as the surfaces being compared for collisions become more complex [7]. Because physics engines desire accuracy, the cost of these calculations is acceptable on a small scale. However, actual collisions are relatively rare occurrences in a physics simulation, and it quickly becomes computationally expensive to do a detailed collision check every **timestep**.

[2] Rafael de Sousa Rocha and Maria Andréia Formico Rodrigues. "An evaluation of a collision handling system using sphere-trees for plausible rigid body animation". In: *Proceedings of the 2008 ACM symposium on Applied computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 1241–1245. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1363972. URL: http://doi.acm.org/10.1145/1363686.1363972.

DEFINITION 2.3 **Timestep:**
The time between each frame in a simulation. Essentially a measure of how smooth a simulation will run; a shorter timestep will look smoother, but it will also require the computer to check for collisions more frequently.

Constantly checking for exact collisions would be about as useful an activity as constantly checking one's phone to see if someone is calling. Receiving a phone call is a (relatively) rare occurrence, so we have indicators (such as a ringtone) to identify when we should check for a phone call. Following this same mentality, efficient collision detection processes will first check if two objects are near each other before even considering the objects for a potential collision [6]. This simplified take on collision detection is much easier for the computer to handle. It's almost as though each object in the environment has its own personal "bubble". If another object is outside of that bubble, then there is no point in spending more time calculating whether a collision actually occurred. When two objects enter each other's bubbles, the computer will know that more accurate calculations are needed to decide whether a collision has occurred. Using this principle, [6] suggests using three phases of collision detection, each more rigorous than the last: reasonable distance detection (broad phase), bounding box detection (narrow phase), and polygon collision (pixel-by-pixel or polygon-by-polygon). Physics engines that adopt this strategy should be able to perform a very vital and computationally-involved process in much less time without sacrificing accuracy or realism.

Discrete Motion and Continuous Motion

THERE ARE OTHER TRADE-OFFS THAT CAN BE MADE with collision detection. As with many factors in physics engines, these trade-offs force the programmers and animators to choose between speed or accuracy. In [6], two different approaches to collision detection are presented: **discrete motion** and **continuous motion**. It is important to understand that computers perform their physics simulations in units of time called frames. The standard for animation and movies is 24 frames per second; the standard for games is typically 30 frames per second. That may seem like a very small window of time, but a great deal can happen in that $\frac{1}{30}$th of a second. These frames become critical when discrete motion is used. Discrete motion can be thought of as a object "warping" from point A to point B every frame, instead of sliding along the whole distance. Most simulated motion is relatively slow; a bouncing ball will only move a fraction of its length every frame. However, a problem arises with objects that move incredibly fast–bullets, for example–which could quite easily travel more than their own length every second. Because the computer will only update the bullet's location once every frame, the bullet could "warp" right through thin objects without ever actually touching them. This collision error is referred to as **collision tunnelling** . [6] This system of discrete motion is an easy and fast way to calculate motion, but can result in very unrealistic situations. If such performance is unacceptable–as is the case in most animation situations–then an alternative method can be used. Known as continuous collision detection, this alternative involves checking for collisions over the entire path traversed by an object over the past frame [6]. While this process is much more computationally expensive, it is undeniably more accurate and avoids the possibility of collision tunneling.

DEFINITION 2.4 **Discrete Motion:** Every frame, an object's position is updated with no consideration for how it actually got from point A to point B. Can be thought of as the object teleporting to its new location every frame.

DEFINITION 2.5 **Continuous Motion:** Every frame, an object's position is updated while considering what it might have collided with in the meantime. This approach is truer to how objects behave in real life, but requires more work for the computer.

DEFINITION 2.6 **Collision Tunnelling:** In discrete motion, when an object moves so fast that it travels through another object without colliding.

Stability and Settling

IT IS IMPORTANT FOR PHYSICS SIMULATIONS to be realistic. Most of the time, **errors** in the simulation are not desired, because they detract from the realism of the simulation. However, there are some things that a simulation can get away with without disrupting the sense of realism. Simply put, the simulation doesn't need to be perfect to be believable. Taking this principle to heart allows physics simulations to cut corners in certain areas without sacrificing quality. In [7][3], three different types of errors are classified. The first and least serious error is aptly referred to as the "imperceptible" error. Most

DEFINITION 2.7 **Error:** Anything visually unexpected in an animation. Errors are many and varied, and can be classified based on their seriousness. Most errors are undesirable in an animation.

[3] Thomas Y. Yeh et al. "Fool me twice: Exploring and exploiting error tolerance in physics-based animation". In: *ACM Trans. Graph.* 29.1 (Dec. 2009), 5:1–5:11. ISSN: 0730-0301. DOI: 10.1145/1640443.1640448. URL: http://doi.acm.org/10.1145/1640443.1640448.

of the time, this class of errors are acceptable in a simulation and, as mentioned before, allow the developers to save time by cutting corners. This leads to a decision: do you sacrifice speed to have a more accurate simulation, or do you sacrifice an acceptable level of accuracy to have a faster simulation?

There are several avenues by which physics engines can cut corners. For example, the simulation can have a longer timestep. Like reducing the number of pages in a flipbook, this makes the simulation easier and faster to draw, but also has the potential to make the motion of the objects become less fluid if taken to a great extreme. There are also shortcuts that can be taken in collision detection, such as using discrete motion instead of continuous motion. However, it is important to not take so many shortcuts that you allow more serious errors to occur. Errors that are visible but don't damage the simulation are classified as "visible but bounded" [7]. Some simple examples of this kind of error are a ball bouncing slightly into a wall or a pile of objects not settling properly. These errors are noticeable and detract from the realism of the simulation in undesirable ways, but they rarely compromise the simulation. The third and most serious classification of error is the "catastrophic" error. [7] These errors can cause the simulation to become unstable. A catastrophic error might happen if an object falls outside of the "world" being simulated. This is especially disastrous during a game if it happens to the player. Ultimately, a good physics engine should be fast, and therefore is expected to cut corners and allow for small, imperceptible errors. Also, a physics engine will never be able to perfectly mirror reality, so errors are to be expected in any case. But, an engine should never allow catastrophic errors to occur. Visual but bounded errors should be as rare as possible.

Constraints

FINALLY, AN IMPORTANT ASPECT OF PHYSICS ENGINES is the use of **constraints**. Constraints are special limits that can be imposed on two simple objects that are connected together. An elevator is a real-life example. Elevators are only allowed to move up and down because there is a constraint connecting it to the elevator shaft which limits the direction of its motion. Constraints allow groups of rigid bodies (referred to as "actors") to be connected together to form a larger, more complex object. [2] Most physics engines support basic objects like spheres, cubes, and cylinders, but constraints allow these simple objects to work together. For example, a car could be made of two pairs of cylindrical wheels attached together by an axle, and then

An example would be a hinge, which is a constraint between a wall and a door.

connecting the axles together and finally setting a box on top of the whole thing to represent the car body. In this example, the wheels on the axles would be under a "hinge" constraint. These types of constraints allow objects to rotate around only one axis (in this case, the wheels can only roll forwards or backwards). [2] The hinge constraint makes sure this limitation is kept. The two axles would need to be attached to each other using a rigid constraint; in other words, we don't want the axles to be able to move at all. Otherwise, the body of the car could bend. This is obviously a simplistic example, but it's a principle that all physics engines should adhere to. The number and variety of available constraints allow developers to create a huge variety of complex objects to simulate. In [2][4], they use constraints to simulate a forklift. Constraints can be used to simulate organic objects as well. Think about how a head can only rotate along certain axes, or how an elbow only bends in one direction. Even your body obeys constraints!

[4] Shih Chung Kang Jhih Ren Juang Wei Han Hung. "Using game engines for physics-based simulations - A forklift". In: *Electronic Journal of Information Technology in Construction* 16 (), pp. 3–22.

Methodology

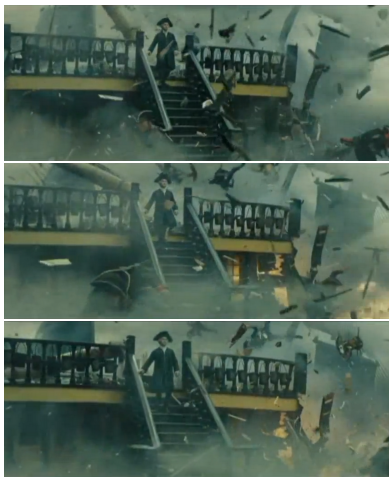

Figure 3.1: Pirates of the Caribbean, End of the Endeavor, first perspective

Figure 3.2: Pirates of the Caribbean, End of the Endeavor, second perspective

To gather empirical data for our study, we devised simple but practical tests that encompassed the needs of animators. Besides the common physics values–namely stablity and accuracy–animators need a consistent **rendering** environment; in other words, the animation needs to play the exact same each time. We tested consistency by duplicating tests and running them simultaneously and isolated from one another. Animations also require complex simulations that involve hundreds if not tens of thousands of objects moving simultaneously. A great example of this requirement occurs in films like Pirates of the Caribbean 3: At World's End, in the scene "The End of the Endeavor" as depicted in Figure 3.1 and 3.2.

Though we were not working with simulations as sophisticated as a pirate ship being blown apart by cannonballs, we tried to imitate the same demanding load in our simulations by running multiple instances of the same experiment in the same environment.

Below, we outline the different experiments we performed and what we hoped to show from them.

definition 3.1 **Render:**
The final process of creating the actual 2D animation from the prepared scene. The end result of the rendering process is the video or image displayed on the screen..

Stack of Rubble

This experiment tests a physic engine's ability to stabilize a pile of rubble. Streams of balls and squares are dropped from a height and left to pile up on the ground. The experiment tests two qualities: the changing state of an object (in this case, shifting between falling/settling and being perfectly motionless) and the visibility of errors. Ideally, the rubble stack should settle quickly and all the objects should reach a motionless state. Physics engines that take shortcuts when attempting to stabilize objects will have unstable stacks in which the objects appear to jiggle very small amounts. This is a minor error that is a side effect of none of the objects in the stack coming to a complete stop. This test will also try the physics engines' abilities to simulate a large quantity of objects without slowing down

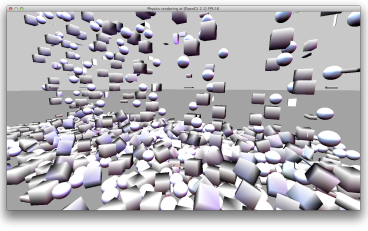

Figure 3.3: Random objects falling in a large stack to test speed performance.

due to an excessive number of collisions. This test is a prototype of a scene in Pixar's *Toy Story 3*, where there are many randomly objects in a pile.

Cannonball

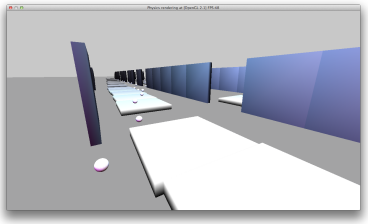

Figure 3.4: Cannon balls launched at increasing speeds, showing where one ball skipped through a wall because of collision detection errors.

This experiment is specifically intended to test the realism of each engine's **timestep** system, as well as the collision paradigm the engine uses. Recall from page 4 the difference between discrete and continuous motion. Engines that use discrete motion could result in the cannonballs passing straight through a wall at high enough speeds. In the cannonball experiment, many cannonballs are shot at varying speeds at a row of walls. The expected behavior is for the cannonballs to collide with the walls and knock them down. Any walls left standing are a clear visible indicator that the cannonball tunneled through the wall.

Stairs

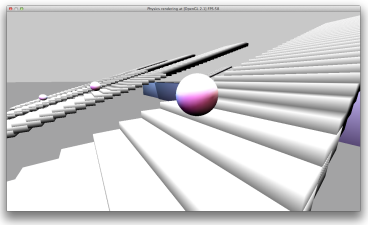

Figure 3.5: Balls rolling down stairs to test consistency.

Balls rolling down stairs can experience a change in constraint as they shift from rolling to falling. This experiment is partially designed to text how each engine handles these constraints. Several rows of stairs are set up, and a ball is dropped from the top of the stairs. This test provides an appropriate circumstance to repeat the same experiment several times within the same environment. This allows us to ensure that the same experiment will yield the exact same results each time it is ran. As mentioned at the beginning of this section, this replication of results is another desirable trait of physics engines.

Implementation

The program running our experiments consists of a simple loop where the physics world proceeds at a fixed timestep. The experiments are set up identically for each physics engine. The engines then take turns running the experiment. A graphical interface is synchronized with the physics world on each frame which allows the user to view the ongoing simulation. This program has an interactive camera which can navigate the scene. Thus, our information about each test was gathered through visual means.

Software components have been created that allow us to create tests easily and consistently across physics engines. This consistency allowed us to make a proper comparison between the three engines we tested. All tests were designed to be agnostic to which physics engine was being used. Some components allowed us to synchronize with a graphics system, such as the open source rendering engine, Irrlicht[1]. With Irrlicht, we could view the simulation from any angle while developing tests. This allowed us to observe qualities such as settling and instability.

[1] http://irrlicht.sourceforge.net/

Results

In this section, we present the observed results of each of the experiments for each engine. It is important to note that the results we gathered have all come from observations as opposed to calculations. Because we are testing physics engines which are intended to produce realistic results, only the errors that are visible are important in considering where one engine excels and another lacks, so it isn't practical to use calculations to determine the results of our experiments.

Stack of Rubble

In this test, we created assorted cubes, spheres, and small randomly-sized rectangular shards to test for object settling and performance. With Bullet we had moderate, but not real-time, speed. The pile fell together and seemed visually realistic. PhysX ran faster than Bullet. At first, PhysX seemed to handle this test very well. However, this test soon revealed some instabilities in how PhysX handles settling. We observed that the pile was actively shifting around and that some objects would be catapulted from the pile into the air. ODE crashed within seconds of running this test, bearing inconclusive results for the qualities we tested for. With some inspection and trial and error, we found that ODE does not handle large clusters of items very well and does not have robust mathematical calculations. It is very ill-suited for this kind of situation.

Cannonball

In this test, we simulated shooting multiple cannonballs, each with increasing linear velocity. We tested for collision detection and collision tunneling. We observed with Bullet that tunneling does occur. However, it had the least serious tunneling of the three engines we tested. We also observed that, across simulations, PhysX and ODE lacked **deterministic** quality. With ODE, we observed another un-

definition 4.1 **Determinism:** The attribute of being predictable. A deterministic simulation can be repeated without any variation.

stable attribute. We saw that the walls that did get knocked down would vibrate on the ground in place. This is a fairly major error hinting at instability within the way ODE handles object settling. See Figure 4.1 on Page 15.

Stairs

We used a ball bouncing down a flight of stairs to test realistic bouncing and rolling constraints. Several flights of stairs and several balls were dropped each test, allowing ease of visual comparison between the different ball/stair sets. Each ball fell at a random velocity. Bullet and PhysX presented believable simulations. Bullet was consistent between simulations, both within one experiment and across several simulations. PhysX was only consistent within individual experiments. With ODE, we observed unbelievable results. One such case was a ball that would fall through an infinitely small crack between stairs and roll around underneath and then fall off from under the stairs after swinging for a few seconds. None of the other engines produced such erratic results.

Summary

OUR EXPERIMENTS WERE DESIGNED TO TEST EACH OF THE ENGINES in five key qualities: collision detection, changing states, discrete or continuous motion, stability or error allowance, and constraints. By paying attention to these qualities, we can report on the consistency, realism, and overall performance of each engine. In the next section, these are the main areas we focus on in our discussion of which engines perform the best in a video animation setting.

We also find that, unsurprisingly, an increase of objects to simulate slows the simulation down. Bullet will maintain its timesteps for the simulation, regardless of the computation speed of the machine it executes on. This means that, if the simulation is becoming too taxing for the computer, it will slow down to compensate instead of reducing the timestep. This would be like a movie playing in slow motion. PhysX and ODE, on the other hand, do not maintain the fixed timesteps if the load is too high. Instead, these engines slow down the timestep. This would be like reducing the frames per second of a movie. This causes occurrence of collision tunneling to skyrocket. The objects often would fall half through each other and then pop back out, creating a visually unappealing scene.

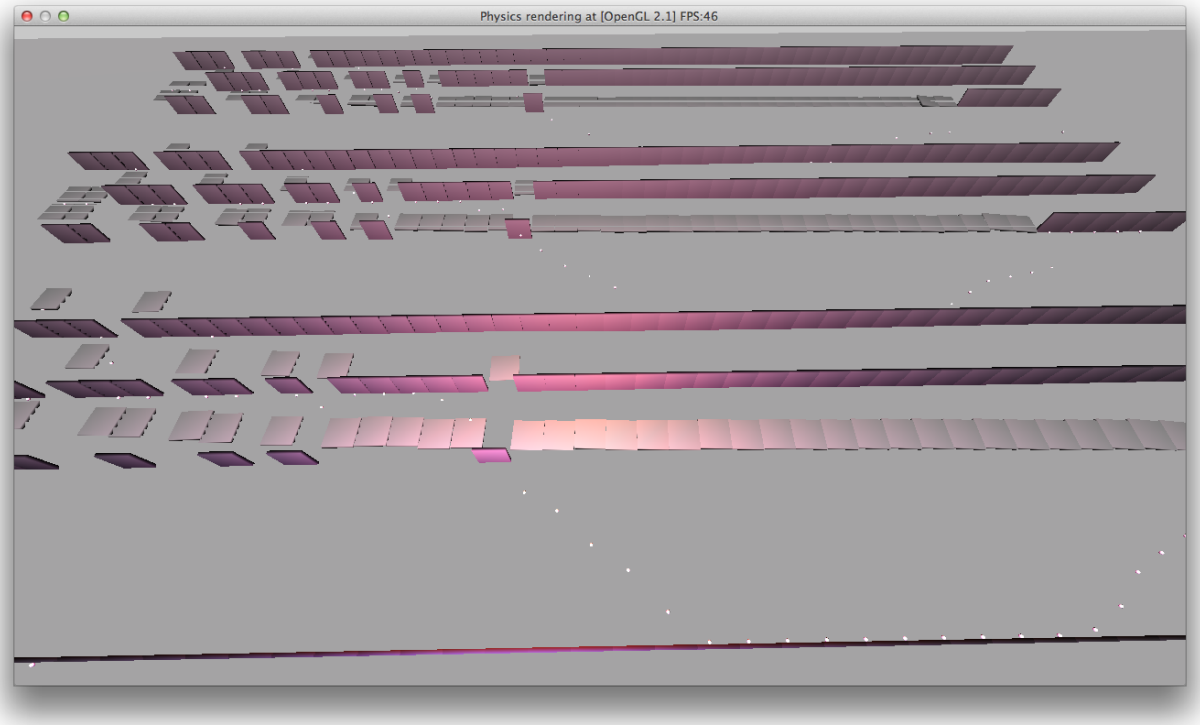

Figure 4.1: This screenshot from our cannon ball experiment shows how we set up three instances of the experiment within one simulation. Each set of three rows of walls is its own experiment. Notice how all three experiments have the same wall standing or knocked down. This shows how repeated simulations have the same results.

Discussion

Physics Engine Performance

Each of the engines we tested have specific strengths and weaknesses that express themselves in different situations. These strengths make the engines better suited to different kinds of animation. For film animation, realism and consistency are the desired attributes. Games, on the other hand, desire speed; realism is a luxury that can be sacrificed to provide a faster simulation. Because animations for film are *pre-rendered*–that is, all of the calculations and simulations are done before the movie is ever viewed–it is much less important for simulations to be super fast. Real-time rendering for movie animation is simply not a priority. Therefore, an engine that is slower but more accurate would be better suited for use in the film industry, whereas one that cuts appropriate corners and **renders** very quickly would be more appropriate for video games.

Another important but less visible feature of a physics engine is its customization and documentation. If animators wish to have a wide range of customization, they need to have developers couple the tools they use with the simulation software, which requires documentation.

After implementing and evaluating Bullet Physics, Nvidia PhysX, and Open Dynamics Engine(ODE), we find that each scored differently in the values we've specified. We rated the engines on each value on a scale from 1 to 10, with 10 being the greatest, see Figure 4.2.

Quality	Bullet	PhysX	ODE
Consistency	10	4	5
Stability	7	5	4
Realism	10	6	3
Speed	6	10	2
Customization	9	8	5
Documentation	7	10	3

Figure 4.2: Table comparison of physics simulators in respect to desired qualities

We found that Bullet Physics rates best in the highest-priority qualities that movie animators desire. Although PhysX may be adequate for real-time simulations, its consistency score makes it not feasible for rendering movies where speed is not as much of a factor. PhysX would be much better suited as a video game physics engine.

Future Research

There is a lot of potential research that could continue from where we have left off. In the investigation of which physics engines perform the best for certain situations, it would be useful to expand the scope to include more of the common physics engines than the three that we studied. Additionally, more experiments could be ran in order to gain a more thorough understanding of the strengths and weaknesses of each engine.

In this paper, we only investigated a few aspects of physics simulation as they apply to **rigid bodies**. While rigid bodies are an important part of animation, there are many more advanced objects that are becoming more and more prevalent in today's movies and video games. Cloth simulation is an active field of research right now. It covers a range of topics from the wrinkles in clothing to the flow of a character's hair. These processes are even more tedious to simulate by hand than rigid bodies because of the incredibly high flexibility of the objects. Another active area of research is in fluid dynamics and the simulation of water surfaces. This is an especially challenging area of research because water is rarely motionless. Unlike cloth, water is not cohesive and is capable of flowing around obstacles and then coming back together on the other side. These two areas of physics simulation both fall into the category of "soft bodies".

DEFINITION 4.2 **Ridgid Body:** A simple, solid object that is capable of moving and rotating but not changing its shape.

Soft body experimentation can follow a similar procedure to the one we follow in this paper. Like rigid body physics, soft bodies must be able to change state, efficiently identify collisions, flow either continuously or discretely, be stable and capable of dealing with errors, and often face motion constraints. Using these five qualities, different physics engines could be analyzed for their ability to effectively and realistically handle soft-body simulation.

Future experiments with rigid body physics may consist of the following.

- Newton's Cradle
- A Box on an incline
- An unbalanced teeter-totter with varying weights

- A box on a spinning platform
- Collisions between wheeled objects
- Plinko simulator
- Sharp edges and a falling ball
- A ladder leaning against a wall

Conclusion

Through a few straightforward experiments, we have observed that, of the physics engines Bullet, PhysX, and ODE, Bullet is the best suited for use in film animation because of its realism. While it may be slower than the second best engine, PhysX, the lack of speed is not a major consideration because of the pre-rendered nature of movies.

In overview, a physics engine is the system that makes computerized objects look like they are behaving in real space and are being affected by real forces. It is important for physics engines to be mostly accurate and quick enough to accomplish the animators' needs. As physics engines continue to improve, we will be presented with more and more realistic animations–be they video games or movies. Truly, physics engines are key in producing today's top entertainment products.

Glossary

Closed Source Programs whose code is not accessible to the general public. Closed-source programs are written by companies or small groups and are often used to make a profit. Often, closed-source programs have a more "professional" appearance and extra features when compared to their open-source counterparts. Google Chrome is an example of a closed-source program that is available for free. 10

Collision Detection Determining whether two objects are touching and calculating their reactions. 9, 14

Collision Tunnelling In descrete motion, when an object moves so fast that it travels through another object without colliding. 15

Constraint Restrictions imposed on an object by preventing motion along certain axes, as with a ball only allowed to roll instead of bounce or a wheel only allowed to spin in one direction on an axle. 13

Continuous Motion Every frame, an object's position is updated while considering what it might have collided with in the meantime. This approach is truer to how objects behave in real life, but requires more work for the computer. 15

Determinism The attribute of being predictable. A deterministic simulation can be repeated without any variation. 23

Discrete Motion Every frame, an object's position is updated with no consideration for how it actually got from point A to point B. Can be thought of as the object teleporting to its new location every frame. 15

Error Anything visually unexpected in an animation. Errors are many and varied, and can be classified based on their seriousness. Most errors are undesirable in an animation. 15

Frame A single snapshot, like a photo of the objects in motion. Several frames viewed in one second create the illusion of motion, much like a flipbook. A single frame represents the snapshot of all the motion that happened in a very short period of time. 9, 15

Graphics Card An essential piece of hardware that performs many of the intense calculations required to determine what is drawn to the screen and where it will be drawn. 10

Open Source Programs whose code is accessible to the general public. Typically, open-source programs are produced as a community effort rather than by a specific company, and–unlike most closed-source programs–are available for free. Mozilla Firefox is an example of an open-source program. 10

Physics Engine A computer program that uses well-known physics equations to calculate the positions and motions of several objects in a simulated world. Physics engines allow users to create an imitation of a real-life scenario to predict how real objects would behave. 9

Render The final process of creating the actual 2D animation from the prepared scene. The end result of the rendering process is the video or image displayed on the screen.. 19, 27

Ridgid Body A simple, solid object that is capable of moving and rotating but not changing its shape. 28

Simulation Giving the computer a scenario and telling the computer to make the objects behave as they would under the laws of physics. Running a simulation is like hitting "play" on a movie. 9

State The present condition of an object which the physics engine uses to decide how the object will behave, especially with gravity or friction. States often describe the motion of an object: motionless, rolling, sliding, falling, no gravity. 13

Timestep The time between each frame in a simulation. Essentially a measure of how smooth a simulation will run; a shorter timestep will look smoother, but it will also require the computer to check for collisions more frequently. 14, 20

Bibliography

[1] Thomas Ellman. "Specification and synthesis of hybrid automata for physics-based animation". English. In: *Automated Software Engineering* 13 (3 2006), pp. 395–418. ISSN: 0928-8910. DOI: 10.1007/s10851-006-8532-4. URL: http://dx.doi.org/10.1007/s10851-006-8532-4.

[2] Shih Chung Kang Jhih Ren Juang Wei Han Hung. "Using game engines for physics-based simulations - A forklift". In: *Electronic Journal of Information Technology in Construction* 16 (), pp. 3–22.

[3] "PDI Dreamworks Megamind, Shrek 4 and 'How to train your dragon' are using Bullet". In: (Oct. 2010). URL: http://bulletphysics.org/wordpress/?p=241.

[4] *PhysX Games*. 2013. URL: https://developer.nvidia.com/physx-games.

[5] *Products that use ODE*. URL: http://ode-wiki.org/wiki/index.php?title=Products_that_use_ODE.

[6] Rafael de Sousa Rocha and Maria Andréia Formico Rodrigues. "An evaluation of a collision handling system using sphere-trees for plausible rigid body animation". In: *Proceedings of the 2008 ACM symposium on Applied computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 1241–1245. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1363972. URL: http://doi.acm.org/10.1145/1363686.1363972.

[7] Thomas Y. Yeh et al. "Fool me twice: Exploring and exploiting error tolerance in physics-based animation". In: *ACM Trans. Graph.* 29.1 (Dec. 2009), 5:1–5:11. ISSN: 0730-0301. DOI: 10.1145/1640443.1640448. URL: http://doi.acm.org/10.1145/1640443.1640448.