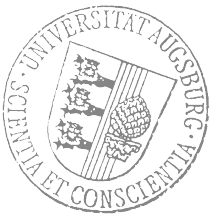
University of Augsburg
Faculty of Applied Computer Science
Department of Computer Science
Bachelor's Program in Computer Science



Bachelor's Thesis

Physics Simulation with Virtual Object and Real Objects on the Microsoft Surface

submitted by
Hassan Aly Selim
on October 2011

Supervisor:
Prof. Dr. Elisabeth André

Adviser:
Mr. Chi-Tai Dang

Abstract

With the rise multi-touch devices, developers and designers started making use of the new multi-touch technology in various applications and games which can be controlled by one or more fingers to achieve various tasks, but until now most of the multi-touch interaction is based on scripted gesture systems. This thesis attempts to take this to the next level by making use of modern physics engines that are used in console and computer games so that the interaction is based on realistic physical interactions similar to real life physical manipulation of objects.
To achieve this a technique was developed which captures the shape of the objects placed on the Microsoft Surface and inserts them into a physics engine that does all the required calculations to make the interaction between the objects feel natural and realistic.
In order to demonstrate what this technique offers, a simple arcade-style base defense game was developed where players try to destroy the opponents' bases by shooting projectiles at it, and players can defend themselves from incoming projectiles by deflecting them using their own hand or any other real object that can be placed on the surface. A lot of Game Design rules and guidelines were taken into consideration to make this game fun and capable of being used in scenarios where there is a multi-touch device placed in a public venue, so people passing by can figure out how to play the game in a matter of seconds without the need of any textual instructions.

Acknowledgments

I would like to thank my Supervisors Mr. Chi-Tai Dang and Prof. Elizabeth André for their constant feedback on my project and for the nice ideas they provided me during the meetings.
I would also like to thank my colleges for helping me with testing the game which made it easier for me find the bugs and glitches in my implementation.

Statement

I hereby confirm that this thesis is my original work towards the Bachelor Degree

Contents

Chapter 1

Introduction

1.1 Motivation

To make interaction with multi-touch surface more immersive by utilizing a physics engine like the ones used in PC and Console games.

1.2 Objectives

The objective of this project was to make a multi-touch game that is fun to play in groups and offers an immersive experience, the game mechanics should be trivial and easy to understand with minimal text instructions.

This was to be achieved by creating an arcade-style base defense game suitable for all kind of gamers, where players shoot projectiles from cannons that destroy the enemies base, and players can block the incoming projectiles by putting their hand (or any object) on the surface and the projectiles will simply bounce off in a realistic way.

Chapter 2

Theoretical Background

2.1 Multi-touch Gaming

When Smart Phones became popular (starting from the iPhone in 2007), developers started making use of the new multi-touch Technology in various applications including games. Game Designers started to design games that can be controlled by one or more fingers to achieve various tasks required by the game.

Multi-touch games require a new form of interaction unlike what we're used to in PC Games and Console Games, and thus not all genres of games are possible in a Multi-touch environment, that's why most smart phone games now are either puzzle games or arcade games.

Since smart phones have a limited screen size, most smart phone games are single player games, that's why multi-touch tabletops and surfaces open up a new possibilities for new types of multi-touch Games, like collaborative games where more than one player can play together to achieve certain goals together as well as competitive games where more than one player go head to head and try to reach certain goals before the other player.

A lot of research has been made on the design of multi-touch games and various researchers formulated rules and guidelines to make a multi-touch game that encourages people to play it, and a lot of researchers focused on multi-touch collaborative games.

2.2 Microsoft Surface

The Microsoft Surface was released in 2008. The Microsoft Surface has infrared light emitters; the infrared light gets reflected by what's placed on the Surface then captured by infrared cameras.

The Surface SDK is a managed library created by Microsoft to give developers access to the multi-touch capabilities of the Microsoft Surface and it's the only supported way to develop for it.

The SDK has low-level classes that give you access to the raw image captured by the Surface's infrared Cameras, as well as medium-level classes that give you access to information about the objects touching the screen in the form of "Contacts".

It also has high-level classes that gives you access to "Manipulations" which process a set of Contacts and calculate the resulting Manipulation made by these contacts. The SDK also has classes made for event-based environments like WPF.

You can make use of the SDK in games/applications developed in XNA or WPF.

2.3 Farseer Physics

The Farseer Physics engine is a managed 2D Physics engine library which build on the XNA port for the popular 2D Physics Engine "Box2D" and adds more features like: Continuous Collision Detection, Convex Polygons and Circles, Multiple shapes per body, Collision Groups, Friction, Restitution and algorithms that decompose concave polygons.

2.4 XNA Framework

The XNA Framework is a managed game development framework made by Microsoft, it has classes for handling Graphics, Keyboard/Mouse/Gamepad Input, Sound, Content Management and Xbox Live (and Games for Windows Live) Services. Graphics Rendering is based on DirectX 9.0c. The version used in this project was XNA 3.1.

Chapter 3

Implementation

The game made use of the medium-level classes supplied by the Surface SDK to detect when the user touches a cannon to fire a projectile, and makes use of the SDK's ability to know the orientation of your finger when it's touching the surface to launch the projectile in the direction your pointing at without having to move your finger in that direction.

The game also makes use of the SDK's low-level classes that provide the image captured by the Surface's infrared cameras to know the shape of the irregular objects placed on the surface and supply it to the physics engine so it can make other game object collide realistically with it.

3.1 Code Architecture

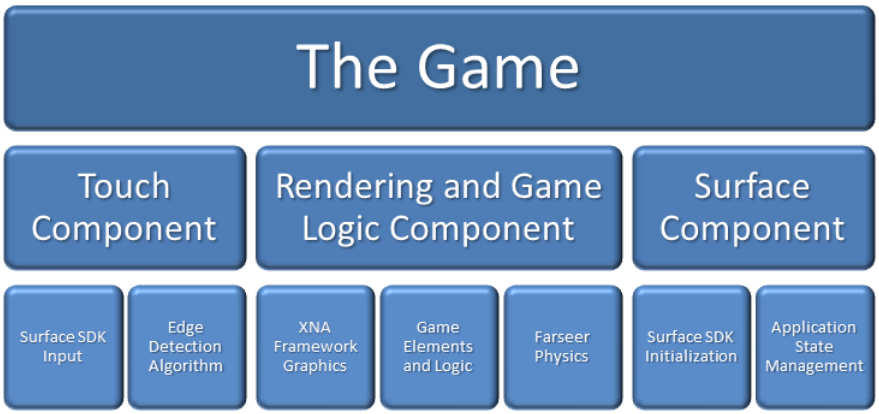


Figure 3.1: General Structure

As seen in figure 3.1, the game is made up of three Components:

- Touch Component: Responsible for retrieving the list of contacts of the current frame, storing the list of contacts of the previous frame and store the raw image capture by the Surface and retrieve it when it's ready. It's also responsible for processing the raw image to extract the Edges of the objects touching the Surface.

It encapsulates all the touch capabilities in the Surface SDK into an XNA Game Component that can be useful in any XNA game that is targeted for the Microsoft Surface.

- Rendering and Game Logic Component: Responsible for Updating and Drawing the current Screen like the Gameplay Screen and the Game Over Screen, and the Gameplay Screen is responsible for managing the Physics World.
- Surface Component: Responsible for initializing the Surface Environment, Display Orientation, and updating the Application State (Application Activated and Application Previewed).

The Gameplay Screen

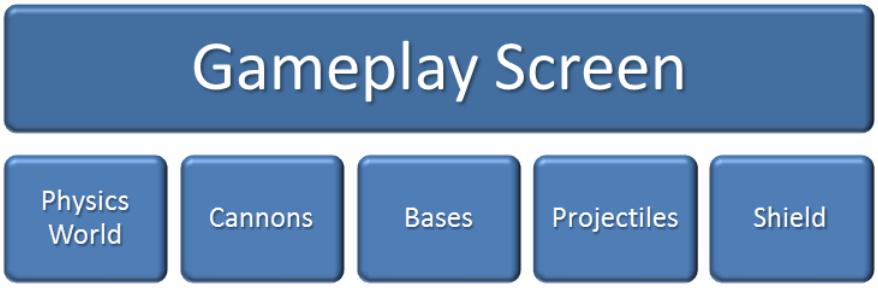


Figure 3.2: General Structure

As seen in figure 3.2, the Gameplay Screen is where everything is combined to make the game, it consists of:

- The Physics World: This is an instance of Farseer's World object; the Gameplay Screen initializes the physics world and inserts in it the game elements that need physical simulation.
- An array of 4 Cannon objects: Responsible for drawing the cannons, detects if a player touches it (and his finger's orientation) and fires a projectile accordingly
- An array of 6 Base objects: Responsible for drawing the bases, detects if it was hit by a projectile, tracks the amount of damage it took, plays the hit and explosion sounds, and initializes the Fire particle effect that happens when a base is destroyed
- An array of 100 Projectile objects: Responsible for drawing all projectiles with their correct color (depending on the owner of the projectile), handles the state of the projectile (e.g.: active/inactive), and draws blurry trails for every projectile
- A Shield object: Responsible for detecting the parts of the external objects placed on the surface that are near the active projectiles (with the help of the Touch Component), it's also responsible for drawing the vertices of these detected parts when in Debug mode

Other Screens

- Start Screen: acts as an overlay at the beginning of the game, it waits until both players touch the screen and then is disappears.
- Game Over Screen: acts as an overlay at the end of the game when a player wins, it shows which of the players won and which one lost, it also displays a nice Fireworks particle effect (along with its sound effect) with the color of the winning player, when the screen is touched the game will restart and show the Start Screen.

3.2 Processing the Raw Input

The aim was to process the raw image captured by the Surface, and to generate polygons that outline the objects touching the Surface, the polygons can later be inserted into the physics world and the old polygons get removed.

The Problems

The main problem that was that the Surface has a relatively weak processor (dual core) and the SDK already uses up a lot of CPU time, and thus doing proper Computer Vision operations was not an option, especially since the game runs at 60 frames per second, so the target was to figure out a way to do this with the smallest CPU overhead possible.

The first step was to extract the vertices for the polygons, and to achieve true vertex detection with irregularly shaped areas is a very complicated and CPU consuming task and thus a simpler way to do this had to be found. Another way was to simply do edge detection and use the resulting edge pixels as vertices for the polygon, this method would give very accurate results but the huge amount of vertices would overload the physics engine, so further filtering was needed to decrease the number of vertices.

The second step was to take the vertices and create one or more polygons that connect these vertices together, and since Farseer doesn't support concave polygon, I had to use a clipping algorithm (Earclip Decomposition) to partition the list of vertices into groups of vertices where each can form a convex polygon, this furthermore emphasis the importance of filtering the pixels that result from the vertex detection algorithm.

To measure the performance I used the .NET class "System.Diagnostics.Stopwatch" to make accurate measurements of the time taken to process the raw image, generate the polygons and add them to Farseer's Physics World.

The solution

At the beginning I used to process the whole raw image every frame but that proved to be very expensive and made it hard to separate multiple objects placed on the surface.

So I improved this by making use of the Surface SDK's high-level Contact classes so that I only process the parts of the raw image that is covered by every Contact, so now I had less processing to do and I could separate different object into separate sets of polygons.

But even after using that approach, the game experienced lags when I place a large irregular object on the surface (e.g.: spreading the palm of my hand on the surface took about 200ms to process), so I ended up processing only the pixels surrounding (and covered by) the projectiles which was reasonable because they were the only objects that are supposed to collide with the objects placed on the surface.

The only problem left was that sometimes there weren't enough vertices detected around the projectiles to give accurate collisions (sometimes the projectiles just passed through with minor decrease in velocity), so I kept tweaking the vertex detection algorithm to give enough vertices without having a huge impact on the performance.

The Algorithm

The initial approach taken was to count the 8 pixels surrounding the current pixel and see how many of them have their intensity higher than a certain intensity threshold, the result of the counting decides whether the current pixel is a vertex or not according to a constant threshold, also the current pixel's intensity had to be below the intensity threshold.

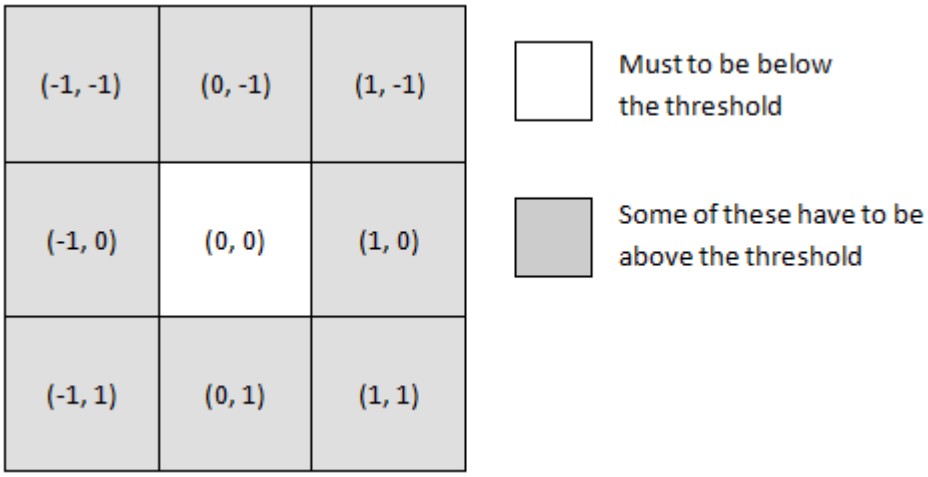


Figure 3.3: First Algorithm

This approach allowed that the count threshold to be changed, this changed the strength of the filtering applied on the edge pixels, and thus indirectly control the number of resulting vertices.

However, the results of that algorithm were sometimes unreliable because it didn't care where the high-intensity pixels are located around the current pixel, and it was slightly slower when compared the second approach.

The second approach taken was to only care about the 4 pixels adjacent to the current pixel, if the intensity one of the adjacent pixels is above the intensity threshold and the current pixel's intensity is below the intensity threshold, then the current pixel is a vertex otherwise it's not.

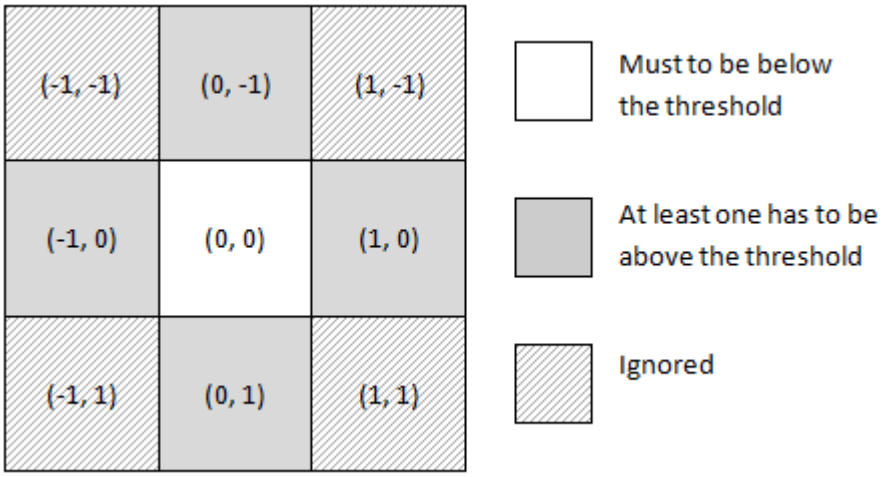


Figure 3.4: Second Algorithm

According to the measurements made, this algorithm was slightly faster (as it just checks the intensity of 4 pixels), and the resulting vertices were more reliable specially after changing the area to process to be just the area around the projectiles.

3.3 Multi-threading

In order to make the game run smoothly without lagging, the Raw Image Processing had to be done on a separate thread, because even at the best of cases the processing (and inserting into the physics world) took about 20 milliseconds and the game runs at 60 fps so every frame has to take 16 ms or less, so even at the best of cases there would be a very significant drop in the frame rate.

The separate thread ran an infinite loop, inside the loop there is another loop that iterates over the projectiles, and for each projectile the 20px by 20px area around it is checked for vertices, then the resulting vertices is passed to an Earclip Decomposer that returns a list of vertex lists where each list represents a convex polygon.

After iterating over all the projectiles, all the vertex lists are used to update the Fixture List of the Shield Body (which is the virtual representation of the real objects placed on the surface).

Then after all that is done the thread is put to sleep, and when the next game frame starts the thread is signaled to resume and do this all over again (because it's in an infinite loop), the main thread doesn't signal the separate thread to resume unless it finished what it's supposed to do, this guarantees that both threads are synced correctly.

While working on the game it was discovered that the Farseer Physics Library isn't thread safe, so doing modifications to the physics world (or one of the Bodies it holds) by more than one thread caused a lot of runtime errors.

So to solve conflicts due to multi-threading, Mutual Exclusiveness had to be preserved when doing one of the following operations:

- Doing the frame Tick (or Step) in the physics world
- Initializing/Activating Projectiles
- Disabling Projectiles (i.e.: excluding them in physics simulation)
- Attaching and Detaching of Fixtures of the Shield Body
- Rendering the vertices of the Shield Body (only in debug mode)
- Initializing a Power-up

This was achieved using the C# "lock" keyword to lock on the instance of the Physics World, which is the common object across all these operations that have the conflict problems.

3.4 The Design

Gameplay Elements

The game was designed to be played either in a 1 vs. 1 style or a 2 vs. 2 style or in any other formation of teams, this was achieved by not putting any constrains or forcing any rules on the players and also by making the gameplay as natural as possible, so any group of people can arrange themselves in any way around the Surface then start by touching the it and in a few seconds they will easily figure out all the game mechanics on their own and start having fun without much learning.

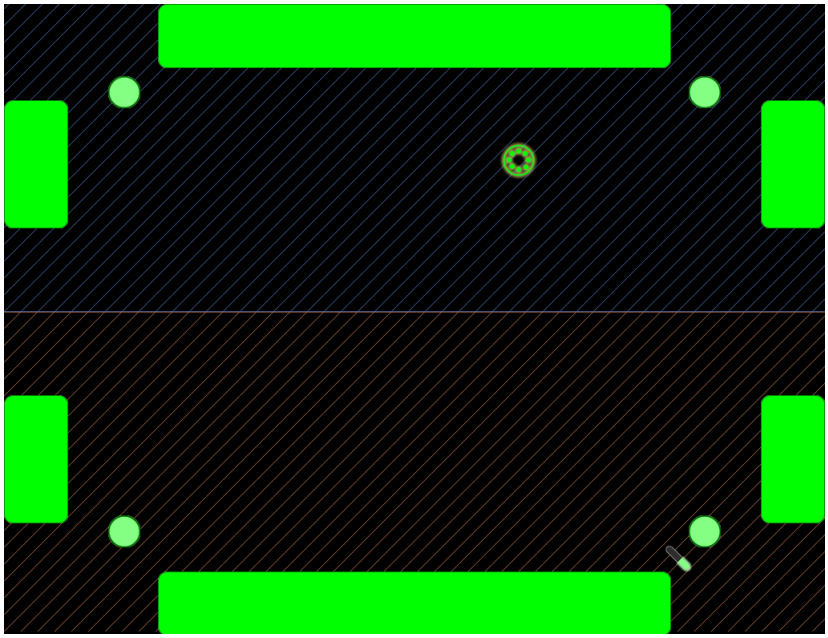


Figure 3.5: Game Screenshot (scaled 40%)

As seen in Figure 3.5, the gameplay elements are as follows:

Cannons

The four circles at the corners are Cannons, when a player touches a cannon, a projectile is launched from it, the longer the player presses the larger and more powerful the projectile is.
The direction of the launched projectile is equal to the average of the last 20 finger orientation values (retrieved from the Surface SDK's Contact class) during the time your finger was touching the Cannon, this makes it very trivial to use the Cannon.

Power Bars

Bar that appears beside the Cannons while touching a Cannon, this shows the power and angle of the projectile that will be launched when player lifts his finger from the Cannon, this makes the projectiles predictable by the player and helps him do more precise shots.

Bases

The rectangles are the players' bases, the long ones can take more damage than the short ones but also they are easier to target and harder to defend of course.
The rectangle changes from Green to Red as the base takes damage and it starts flashing white when it has 2 or less hits left to get destroyed, this color representation is recognized faster than textual representation.
Once destroyed, a fire/destruction particle effect is shown at the base for a short time.

Team Sides

The colored diagonal stripes along with the two horizontal lines at the center shows how the battle ground is split between the Red and Blue teams.
This makes it easy for players to figure out how to arrange themselves around the Surface without any text instructions.

Power-ups

The circle that has other smaller circles inside it is "Scatter Bomb" Power-up that randomly pops up in the battle field, when the Scatter Bomb hit by a projectile it launches 10 projectiles with random angles and random powers, this adds randomness to the game and makes it more fun as smart players will be able to use it in their advantage while others might fail to do so.
The game is designed such that it's easy to later add more types of Power-ups to it. Power-ups where made to activate by hitting them with a projectile and not by touching them to avoid physical clashes between the players' hands.

Auditory and Visual Feedback

One of the aspects of Game Design that the project also focused on was Visual and Auditory Feedback.

Visual Feedback can be seen as the Power Bar that appears next to a Cannon to show the power and direction of the projectile that gets launched, the bases' change in color to show the damage that it took, Particle Effects at the place of collision between the projectile and the base, larger Particle Effects along the length of the base when a base gets completely destroyed and Fireworks Effect when a team wins where the color of the Fireworks is the same as the color of the winning team.

Auditory Feedback was implemented to furthermore enhance the players' understanding of what's going on in the game. Certain sounds were played when a player launches a projectile from a Cannon, when a Base is hit by a projectile, when a Base gets completely destroyed and alongside the Fireworks at the winning screen.

The sounds where tweaked and adjusted so that they are loud enough to be heard yet still not noisy even when there is a lot of things happening in the game, and they were made to be in sync with the Visual feedback as much as possible, all that lead to a full experience when playing the game.

Chapter 4

Conclusion & Future Work

4.1 Conclusion

This project successfully created a base for physical interaction between virtual objects and real objects, that base can be used to implement immersive games that are fun to play and can be extended to make a system that can make complicated physical simulations.

The limitations of doing this on the Microsoft Surface using the official SDK was discovered, and ways to increase the performance the system was demonstrated, further research can be built on these results to furthermore discover the possibilities and uses for such a system.

4.2 Future Work

One of the main problems faced when implementing this project was the Performance Issues, and one of the main causes of this was that I was using a managed programming environment and I had no other choice because Microsoft only provides a managed library for programming on the Microsoft Surface.

So due to this, it's important to try re-creating this work on a different table-top device that has an unmanaged library, that way the image processing operations and the physics simulation would perform much faster.

The performance boost gained from using unmanaged code instead of managed code will allow for applying more accurate vertex detection and more complicated physical simulation.

Chapter 5

References

[1] Microsoft Surface SDK on MSDN Library. http://msdn.microsoft.com/en-us/library/ee804767.aspx

[2] Farseer Physics Engine on Codeplex. http://farseerphysics.codeplex.com/

[3] Andrew D. Wilson, Shahram Izadi, Otmar Hilliges, Armando Garcia-Mendoza, David Kirk. Bringing Physics to the Surface.

[4] J. Leitner, M. Haller, K. Yun, W. Woo, M. Sugimoto, M. Inami, A. D. Cheok, H. D. Been-Lirin. Physical Interfaces for Tabletop Games.

[5] Jonas Schild, Maic Masuch. Game Design for Ad-Hoc Multi-Touch Gameplay on Large Tabletop Displays.

[6] Rilla Khaled, Hannah Johnston, Pippin Barr, Robert Biddle. Let's Clean Up This Mess: Exploring Multi-Touch Collaborative Play.

[7] Alissa N. Antle, Allen Bevans, Josh Tanenbaum, Katie Seaborn, Sije Wang. Futura: Design for Collaborative Learning and Game Play on a Multi-touch Digital Tabletop.

[8] Anne Marie Piper, Eileen O'Brein, Meredith Ringel Morris, Terry Winograd. SIDES: A Cooperative Tabletop Computer Game for Social Development.

List of Figures