“OrbitClash”: My First Video Game

Project Write-up

Author: Justin Weaver
Date: Apr 28th, 2011

Instructor: Kenrick Mock
UAA – Computer Science Department

Table of Contents

Abstract

(top)

OrbitClash is my first attempt at a video game. The project was inspired by the classic game "Spacewar!" It requires the Microsoft .NET 4.0 framework to run. When completed, the game was released freely to the public under an open source license.

1. Introduction

(top)

Whenever I do a project, I always keep a couple of goals in mind: learn something new, and produce quality results. For the purpose of this project, I served as my own client. I am a very motivated client with extremely high standards. So, I didn't make things easy on myself. Being my own client gave me the advantage of total creative control. However, it also left me with no starting direction.

The gaming industry is currently booming, so some rudimentary knowledge of game programming could be handy. With that in mind, I decided to make my first video game!

2. Overview

(top)

When I was young (and dinosaurs roamed the Earth), one of my favorite "casual" games was *Spacewar!*[R1][Illustration 1]

To be completely honest, I don't remember the name of the game, because it was actually just a clone of *Spacewar!* for my beloved Amiga 500. So, embodied in this game, is my attempt to recreate the essence of that childhood memory.

Illustration 1: A Spacewar! clone.

3. Requirements

(top)

3.1 System Requirements

(top)

- The game must run on the Microsoft .NET 4.0[R2] framework.
- The game must accommodate 2-players at one keyboard.
- The game must have "good" graphics, sound effects, and gameplay – as determined by the client. Obviously, this type of evaluation is quite qualitative. However, since I am my own client, I have the advantage of a constant feedback loop. Thus, a qualitative measurement of success is sufficient for the purposes of this project.

3.2 Gameplay Description

(top)

- The game is a top-down, 2-dimensional, non-scrolling, spaceship action/shooter, wherein two players duel to the death!
- Each player controls a small spaceship, which is equipped with a shield (limited), a front-facing cannon (see next point), and two (front and rear-facing) thrusters (unlimited).
- Each ship's cannon has unlimited capacity, with the exception that there can be only 10 "live" bullets from each player in the universe at one time. I imposed this rule, rather than a lengthier

cooldown between shots, to make the game a little more challenging. Note that the player that owns a bullet will never be harmed by it, rather it would simply bounce off them.

- Firing a ship's thruster accelerates the ship in its current direction. When the thruster is not firing, the ship continues on its trajectory unless affected by another force (e.g., collision, gravity, etc). Note that cannon bullets behave essentially the same way.
- The entire game-world consists of the fixed screen area only. Ships and cannon bullets bounce back if they touch the edge of the screen. However, cannon bullets have a limited lifespan, so they disappear fairly quickly.
- At the center of the screen is a planet, which exerts a gravitational pull on the ships and cannon bullets within a specified distance range. Anything that touches the planet is instantly destroyed.

3.3 Program Behavior

(top)

- When the application starts up, the game prompts: "Press the spacebar to begin."
- When the game begins, each ship begins on opposite sides of the screen, with the planet between them. Each ship's starting point is a safe distance from the planet, so that they are outside the range of the planet's gravity.
- When a player is destroyed, they simply respawn after a few seconds in a location that is on the opposite side of (and a minimum safe distance from) the planet from the other player.
- Each player begins with three simple counters displayed:
  - Kills: a count of times the player destroyed other player.
  - Defeats: a count of times the player was destroyed by other player.
  - Suicides: a count of times the player hit the planet.
- The game never ends, it simply goes on until it is shut down.

3.4 Player Controls

(top)

- **Player One Controls (The Yellow Ship):**
  - Rotate-left: A
  - Rotate-right: D
  - Forward-thruster: W
  - Reverse-thruster: S
  - Fire-cannon: Left Control
- **Player Two Controls (The Red Ship):**
  - Rotate-left: Left Arrow
  - Rotate-right: Right Arrow
  - Forward-thruster: Up Arrow
  - Reverse-thruster: Down Arrow
  - Fire-cannon: Right Alt

4. Design

(top)

Since I was completely new to game design, and possessed only a limited understanding of graphics programming, I decided to use a multimedia library called Simple Directmedia Layer[R3] (SDL) to facilitate the game's development. SDL aids game development by simplifying the handling of animation, physics, sound, and player input.

4.1 Simple Directmedia Layer (SDL)

(top)

SDL is written in C, but I used a wrapper called SdlDotNet[R4], which allows much of SDL's functionality to be used from within the Microsoft .NET framework's managed environment. SDL.NET uses .NET's Interop services to access SDL's unmanaged resources, which incurs a small overhead.

SDL is an event-driven framework. Consequently, the game is built from a framework of abstract data types that are manipulated by a single main game class (or controller) in response to events generated by SDL.

4.1.1 Surfaces, Sprites, and Animated Sprites

(top)

A Sprite is a two dimensional image (or Surface), often with a transparent background, that can be quickly pasted into another Surface. An animated sprite is designed, not only to move around the screen, but also to simultaneously play back a kind of flip-book of sprites, which are encapsulated in a sprite sheet. A sprite sheet is really just a series of statically-sized, sequential, motion-capturing frames, not unlike a strip of film.

During gameplay, the ships in OrbitClash will not just move around the screen; they will also be able to rotate. So, a sprite sheet was required to animate the rotation of each ship. Other options were explored, as I will discuss later, but in the end I used sprite sheets.

4.1.2 Events

(top)

Once the initial environment has been set up, SDL is an event driven framework. A framerate is assigned, and SDL calls your own Tick handler for each refresh. Additionally, other handlers can be called in response to events like keypresses, or application close. The programmer chooses which events to attend and how to handle them.

OrbitClash's keypress event handlers merely set flags, which are later read and processed in the Tick handler.

Most of the action takes place in the Tick handler, which updates the states of all the particles in the universe and then draws them to the screen.

4.1.3 Particles

(top)

SDL.NET (note: the wrapper) provides a particle engine that allows an instance of the Particle class to be created from a variety of different objects (like a Sprite, for example).

The ParticleSystem class consists of a list of particles and a list of particle manipulators. A

particle manipulator is a class that implements the IParticleManipulator interface, which forces the programmer to provide an operation (Manipulate) that takes a list of particles and performs some arbitrary task on them. Particles behave according to their Velocity property. The Velocity property contains an abstract data type called a Vector, which is composed of a 3-dimensional origin point, a direction, and a length.

In the Tick handler, the ParticleSystem's Update and Refresh operations are called to send Particles flying around the screen. A call to Update runs each particle manipulator in the system on every Particle in the system, and updates each particle's position according to its Velocity. Each particle has a Life property that ticks down incrementally each time Update is called. If Life reaches 0, the Particle is pruned from the system. However, if the lifespan is set to a negative value, the particle lives forever (or until it is killed by manually setting its Life to 0).

The ParticleEmitter class, (which is also just a Particle itself) emits a steady stream of particles in a configurable way. The particle emitter was very handy for various important special effects. I used particle pixel emitters for the thrusters of each ship's engine, and particle circle emitters for the ship-warping-in and ship-explosion effect.

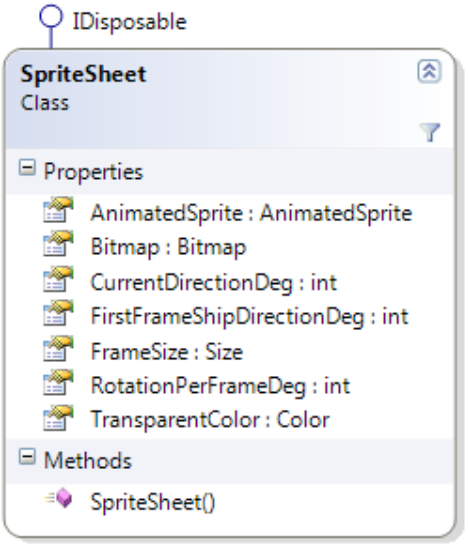
4.1.4 Sounds (top)

Sounds are quite simple to use in SDL.NET. Each sound is loaded from an OGG Vorbis file (other formats are supported, e.g., MP3, WAV) during the program's initialization. The Thruster class holds the sounds for each ship's thruster. The Cannon class carries the fire and dryfire sounds. Since it handles all of the collision detection and reaction, the main game class itself (OrbitClash) holds the collision sounds. Asynchronous playback of each sound is initiated from the Tick handler.

4.1.5 Fonts / Text (top)

SDL.NET provides simple classes that are used to facilitate the loading of a font from a file, and the rendering of a text string to a Surface. The details are of little interest.

4.2 Data Structures (top)



4.2.1 The Configuration Static Class (top)

The configuration class contains many intricate settings OrbitClash, some of which are described in this document. However, if you desire more specific details, please see the comments in the Configuration.cs file.

4.2.2 The SpriteSheet Class (top)

The SpriteSheet class is designed to hold information

about a ship rotation animation. Its constructor takes, among other things, the filename of the sprite sheet image to load as a parameter. The class provides access to the entire image, as well as configuration information like: what direction the ship is facing in the first frame, how many degrees the ship rotates in each frame, the size of each frame in pixels, and the color that will be treated as transparent.

Some of the information the class provides is dynamic. For example, the CurrentDirectionDeg property provides the current direction that the ship is facing by multiplying the degrees of rotation per frame by the index number of the current animation frame.

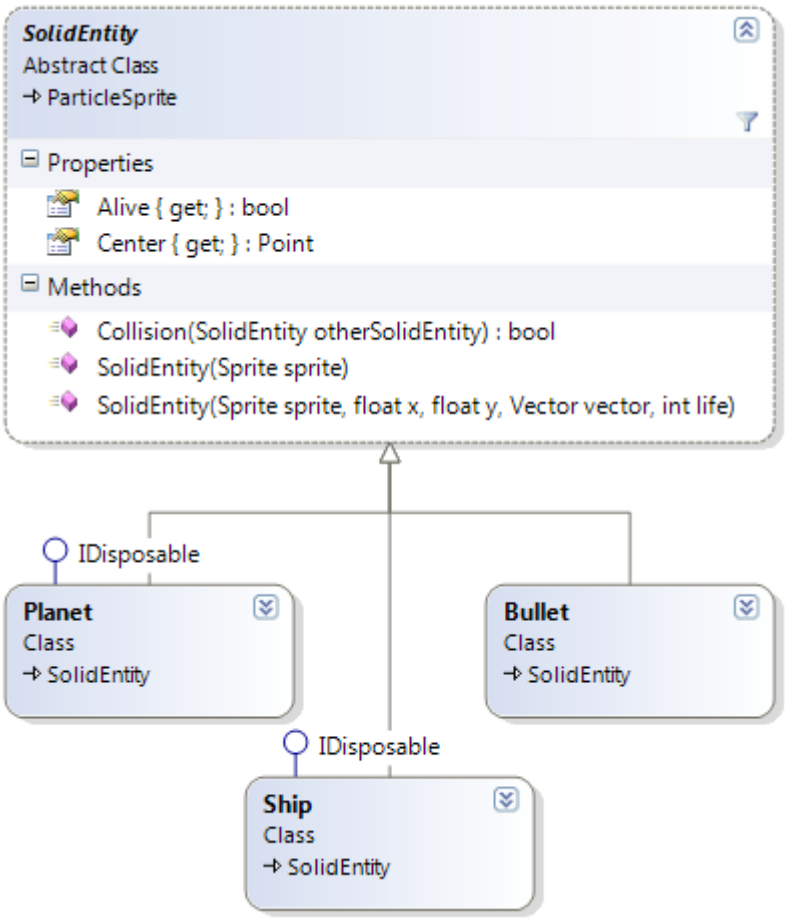
4.2.3 The SolidEntity Class

(top)

The SolidEntity abstract class extends the ParticleSprite class, and is designed to facilitate collision detection between classes that further extend it.

The Collision operation returns true if a pixel-level collision is detected between the entity in question and the entity specified as a parameter.

In OrbitClash, the Planet, Ship, and Bullet classes all extend the SolidEntity abstract class.



4.2.4 The Ship, Cannon, and Thruster Classes

(top)

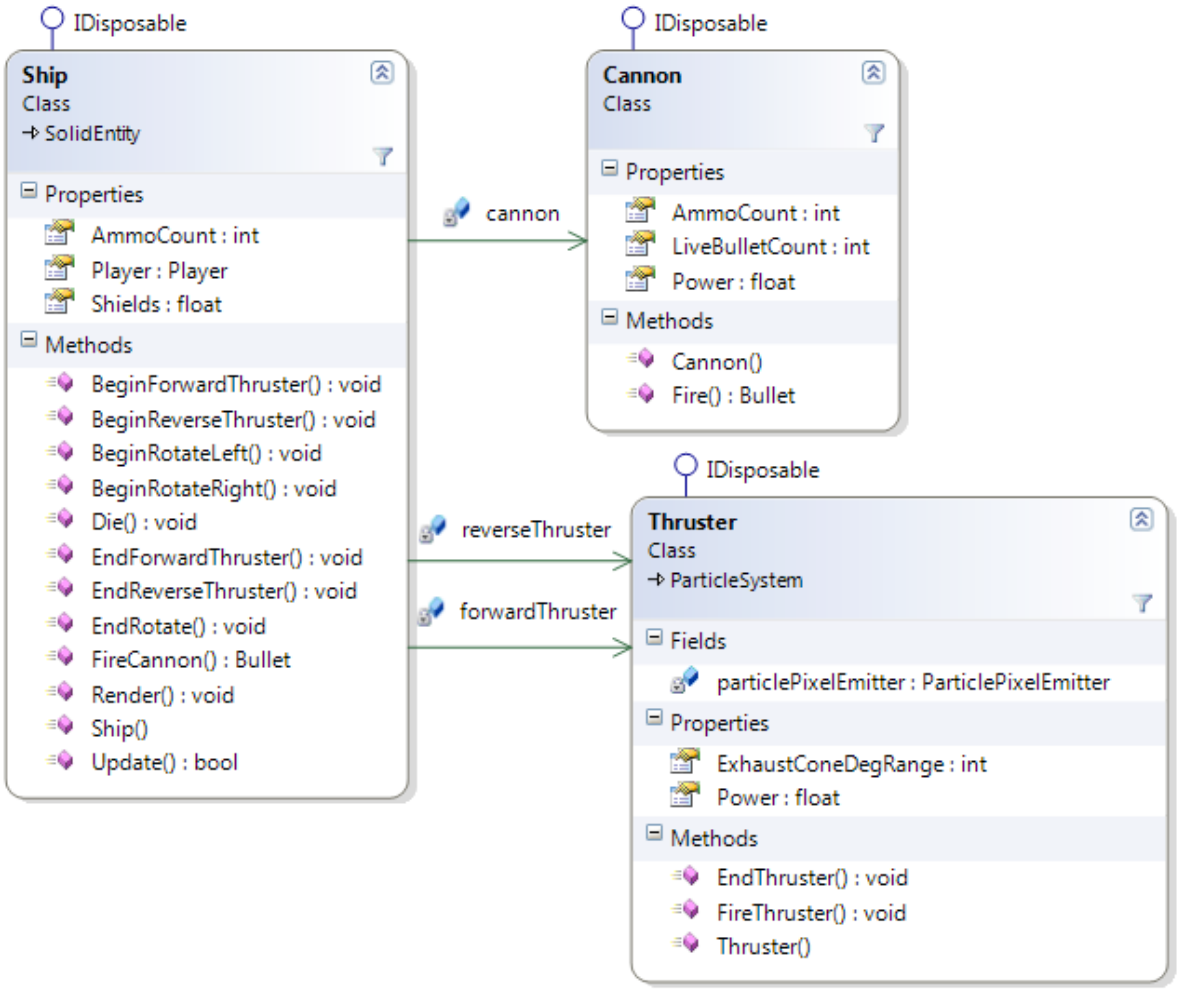
The ship class extends the SolidEntity class, and is therefore also a ParticleSprite. It carries with it: two Thruster class instances (forward and reverse), and a Cannon instance. It abstracts operations on the thrusters and cannon by keeping those data structures private, and passing some requests directly to them (e.g., the AmmoCount property).

The Ship's Die operation causes the ship to turn off its thrusters, disable its weapons, and set it's particle's Life property to 0; this effectively makes the ship hidden and inactive.

The BeginRotateLeft, BeginRotateRight, and EndRotate operations control the ship's sprite sheet animation.

The Thruster class extends the ParticleSystem class, and contains a ParticlePixelEmitter, which creates the engine exhaust effect. By not including the engine effects in the main particle system of the OrbitClash class, I keep them from being effected by the manipulators attached with that system (i.e., gravity, screen edge bounce, speed limit).

The Ship class provides Update and Render methods,which are called from within OrbitClash's Tick handler. Those methods simply call the operations by the same name within the Thruster class instances. Thus, the particles are all updated and rendered, even though they do not exist in the game's main particle system.



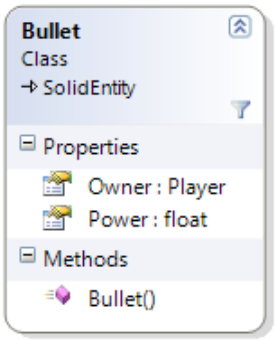
The Cannon class's Fire operation returns a Bullet on success, and null on failure. The cannon may fail to fire if there are too many live bullets for that ship, or if the cannon's cooldown period (as specified in the configuration file) has not elapsed.

To determine the number of live bullets the ship currently has in-play, the Cannon class maintains a list of the live bullets it has fired. It provides the LiveBulletCount property, which gets called each time a player attempts to fire their cannon. When this occurs, first it prunes its list of all bullets with Life of 0, then it returns the count of the number of items remaining in the list.

4.2.5 The Bullet Class

(top)

The Bullet class is a SolidEntity that holds a reference back to the player that owns it. The Power property indicates how much damage the bullet will do if it hits the opposing player's ship. The Bullet class's constructor takes a variety of parameters, including the bullet sprite, the initial bullet Vector, the bullet's Life, and a reference to its owner. The reference to owner is important, because friendly bullets will merely bounce off a ship, while hostile ones cause damage!

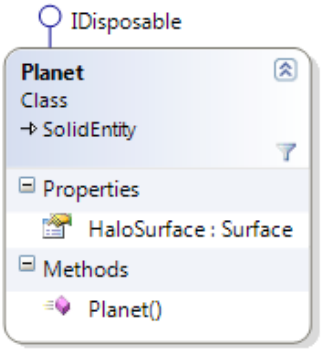


4.2.6 The Planet Class

(top)

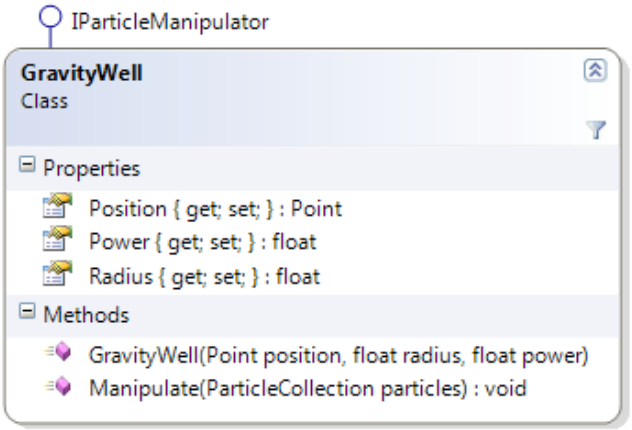
The Planet class is a SolidEntity that, aside from encapsulating the planet image, also contains the HaloSurface property, which is optionally applied to the background surface (i.e., the star field), depending on configuration settings. The halo simply gives a visual representation of the limit of the reach of the gravitational force of the planet.

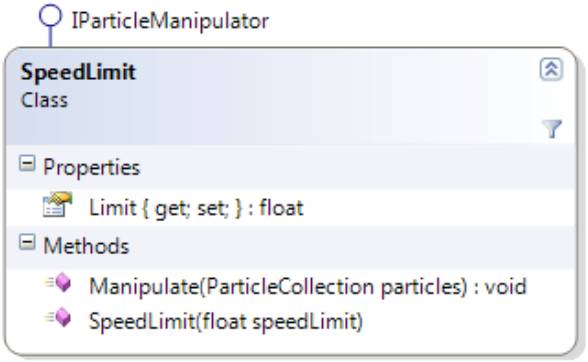
The gravitational force itself is applied via a particle manipulator, which is discussed below.



4.2.7 The SpeedLimit and GravityWell Classes (Particle Manipulators)

(top)





The GravityWell and SpeedLimit classes are particle manipulators that implement SDL.NET's IParticleManipulator interface.

Manipulators are designed to be added to a particle system, which means they run of every particle in the main system each time the main system's Update operation is called.

Optionally, manipulators can be run manually on a collection of particles by calling their Manipulate operation directly.

GravityWell draws all particles within Radius distance of Position according to a calculation that makes the drawing-force zero at the outer edge of the radius, and maximal at the center of the well. The force becomes much more intense as the distance from the center approaches zero. The exact description of the force calculation is: the gravity well's Power divided by the particle's distance from the center of the

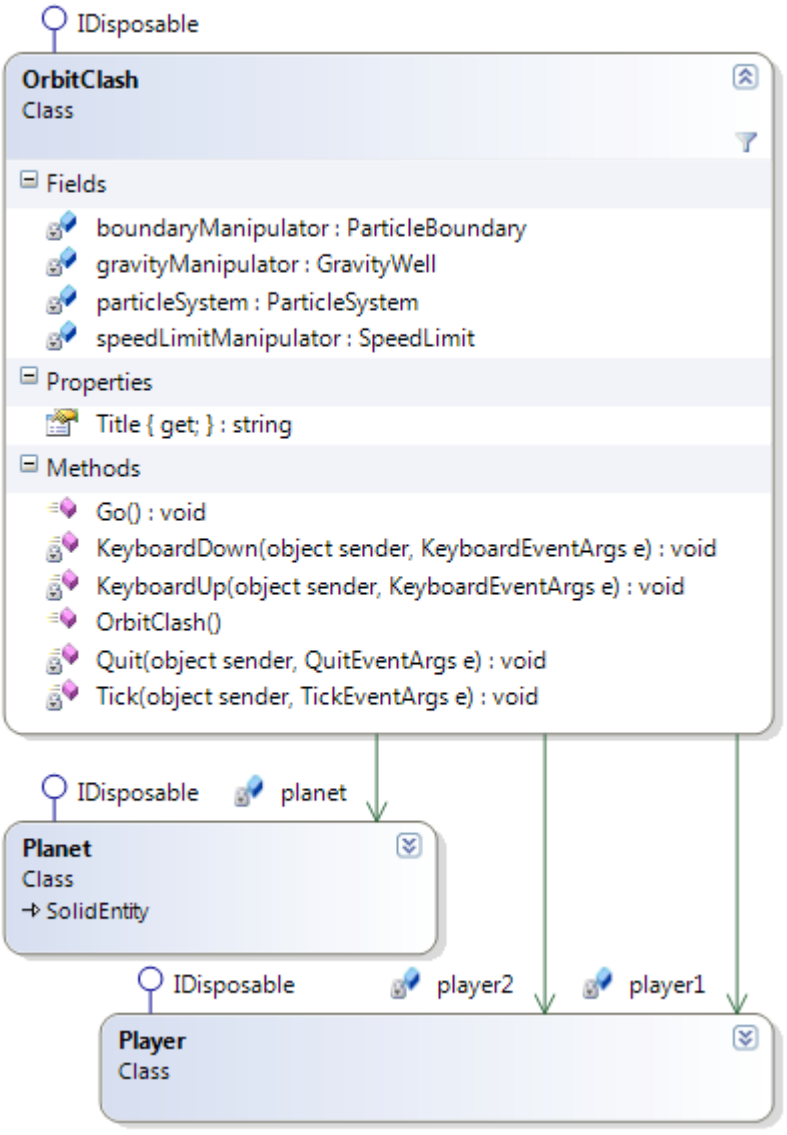
well (as a percentage of the overall well radius) squared. The main OrbitClash class creates an instance of GravityWell and adds it to the main particle system.

SpeedLimit checks and caps the Length property of each particle's Vector. The main OrbitClash class also creates an instance of the SpeedLimit class. However, it enforces the speed limit manipulator manually (as described above), to ensure that it is the last thing to update the particle system before the call to Render.

4.2.8 The OrbitClash Class (and the Tick handler)

(top)

The OrbitClash class is the main class of the program. It handles events generated by SDL.NET (i.e., keypress, tick, quit), maintains the primary game particle system, and does the overseer work of detecting and reacting to collisions.



Naturally, it also contains two instances of the Player class (one for each player), and a single instance of the Planet class.

It is also responsible for instantiating and adding the various particle manipulators used in the game (i.e., boundary, gravity) to the particle system. Note that the speed limit manipulator is not added to the main particle system, but is run manually (as noted above).

Ideally nearly all handling of collision detection and reaction would be better if it were abstracted from the main game class. However, due to time constraints, I decided that making that change – in an already working system – just wasn't worth it.

4.2.8.1 The "Tick" Handler

(top)

In a bare-bones fashion, the Tick handler looks something like this:

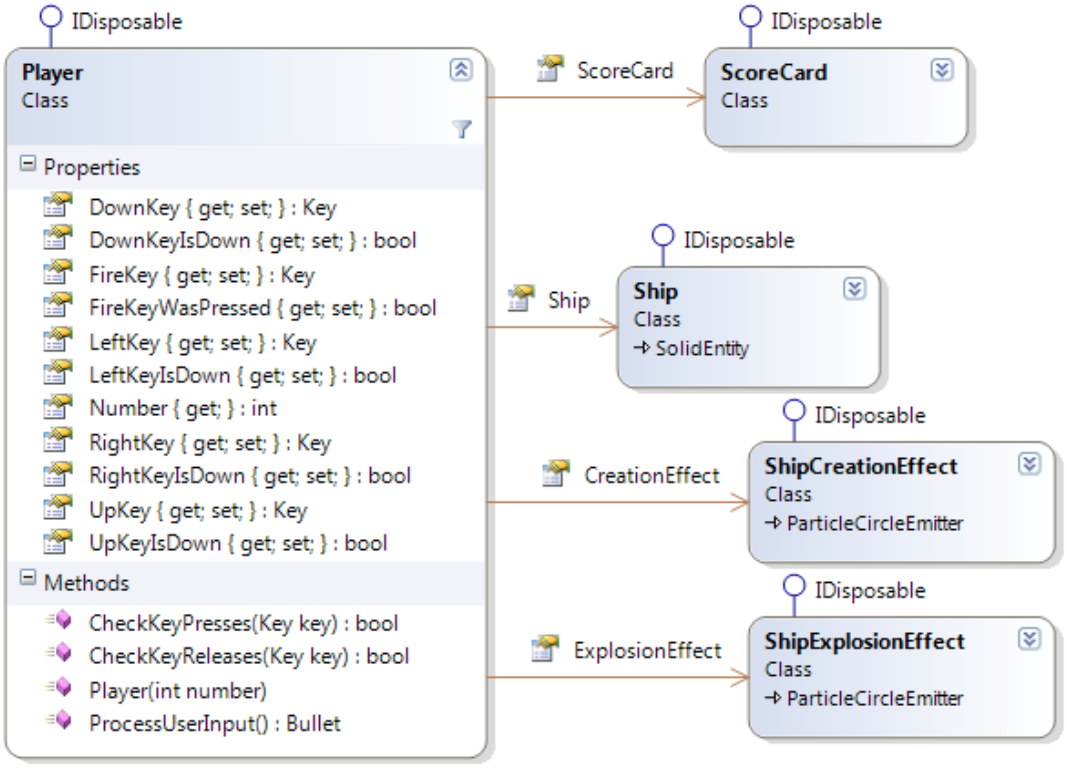
```
/* Update all the particles in the universe by running each manipulator
 * in the system on the list of particles,  and then updating their
 * position coordinates according to their resulting Velocity property.
 */
myParticleSystem.Update();
// Check for, and handle, user input.
ProcessUserInput(); // Local method.
// Check for, and enforce, collisions.
EnforceCollisions(); // Local method.
// Manually enforce speed limit manipulator, to make sure it runs last.
speedLimiter.Manipulate(myParticleSystem.Particles);
// Render all the particles in the universe to the back buffer.
myParticleSystem.Render(Video.Screen);
// Blit the InfoBar at the bottom of the screen to the back buffer.
DisplayInfoBar(); // Local method.
// Finally, flip the back-buffer onto the screen.
Video.Screen.Update();
```

4.2.9 The Player Class

(top)

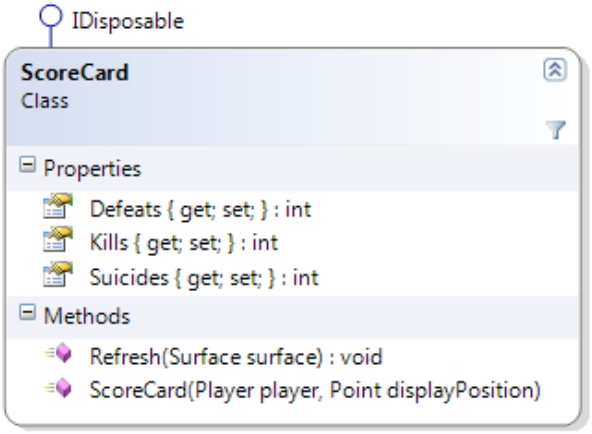
The Player class represents a real-life player, and contains a Ship and ScoreCard instance; it also carries instances of the creation and explosion effect classes.

Its other job, is to take user commands when its CheckKeyPresses and CheckKeyReleases operations are called by the main game class' keypress event handlers, and then execute those commands when the ProcessUserInput operation is called from within the main game class' Tick handler.



4.2.10 The ScoreCard Class

(top)



The ScoreCard class holds statistical information for a Player. Its constructor takes a player and a display position as parameters.

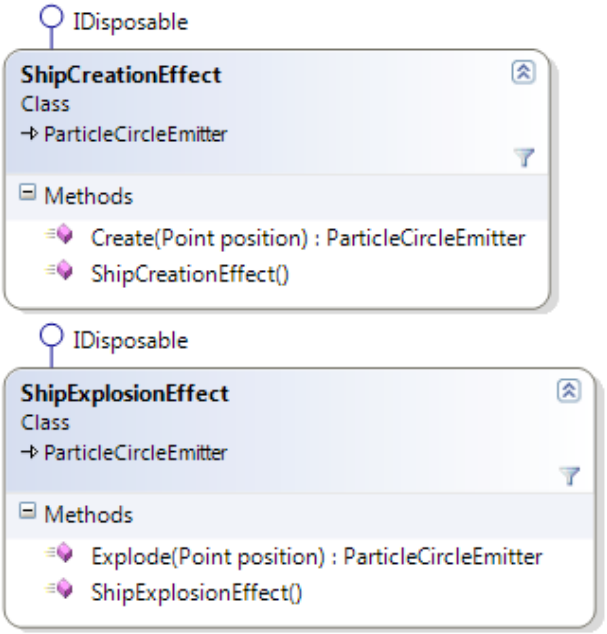
The ScoreCard class will render itself to any surface passed to the Render operation (in keeping with SDL.NET conventions).

Stats Explanation:

- *Kill*: killed the other player with bullets.
- *Defeat*: killed by the other player's bullets.
- *Suicide*: killed by planetary impact

4.2.11 The ShipExplosionEffect and ShipCreationEffect Classes

(top)



The ShipCreationEffect and ShipExplosionEffect classes are very similar, and each is an extension of SDL.NET's ParticleCircleEmitter class.

They each provide a single operation that triggers their functionality at a specified screen position. Each of these classes creates a simple burst of circle-particles as defined in their respective sections of the configuration file.

An instance of each of these classes reside inside, and are utilized by, instances of the Player class.

4.3 Algorithms

(top)

4.3.1 Collision Detection

(top)

One of the most interesting algorithms I wrote is the one that implements collision detection. Collision detection is simply a method of recognizing when two particles come in contact (collide).

One easy method of collision detection is to simply see if the rectangles of the sprites overlap.

However, since most sprites come with transparent portions, the rectangles may not tell the whole story. Although simple rectangle detection may work fine in some scenarios, for the purpose of my game, I needed pixel-level detection.

For pixel-level collision detection, the previously described "rectangle check" is done first; because if the rectangles aren't overlapping, then there is no reason to do the more costly pixel-level check.

If the full pixel check is necessary, then we find the rectangle that contains the intersection of the two sprites, and check each pixel within the rectangle. If we find a location where two non-transparent pixels overlap, then we have a collision.

5. The Development Process

(top)

I chose the name *OrbitClash*, because it seemed appropriate to the game's concept, and more importantly: a Google search for "orbit clash game" didn't reveal any other video games with the same title!

5.1 The Spike Project (A.K.A. The Early Start)

(top)

I really had no idea how to make a game, so I couldn't write a design document without doing a little programming work first. So, to familiarize myself with game programming, I performed a "Spike" project.

My spike resulted in a simple program wherein two very primitive looking ships could be flung around the screen using the keyboard controls. The ships bounced off the edges of the screen, thanks to SDL.NET's ParticleBoundary class, which implements IParticleManipulator and keeps particles within a specified rectangular boundary. However, they still passed right through each other (and the planet), because I hadn't implemented collision detection yet.

The Spike was very helpful because of my inexperience, and it was good quality work. So, I used the results of my spike as my first evolutionary prototype.

5.2 Ship Rotation: Sprite Sheets vs Real Time Rendering

(top)

SDL does not provide an efficient method of rendering the ship rotation in real time. The rotation would be done by the CPU, rather than the graphics hardware. Modern machines are fast, and this would hardly be a killer. However, I wanted to do this "right." So, I was faced with a choice: learn OpenGL and do the rendering on the graphics hardware, or make a sprite sheet and use SDL as I originally planned. I opted to use a sprite sheet for a variety of reasons. Mostly, I did not want to add the extra complexity of learning OpenGL to my already busy agenda.

So, I needed a sprite sheet showing the full 360 degrees of rotation for each ship. I have been informed that automatically creating a sprite sheet of a rotating image is trivial in professional art software (like Adobe Photoshop). Unfortunately, when free alternatives are elusive, poor college students must make our own solutions. So, I whipped up a quick spinning-sprite-sheet-creator of my own in C#.

5.3 Ship Graphics, Sound, and Fonts. (top)

My original plan for making the ship graphics was to draw them by hand, scan them, and shrink them down to hide the defects. However, after a few less-than-spectacular attempts, I located some free ship graphics to use. I edited them for my needs, ran them through my sprite sheet spinner program, and made sure to give attribution to the artist in all deliverable and published materials (including this paper: below).

The planet graphic, sounds, and font that I used are all released under free licenses (as detailed in the *Licensing* section of this paper: below). Everything not mentioned was created by me!

5.4 Testing and Debugging (top)

Since I was my own client, I fell into regular loops of testing, debugging, and refactoring. Along the way, I faced a number of memorable, educational, and even amusing challenges.

5.4.1 Animated Sprites and Collision Detection (top)

The fact that the ship sprites are animated meant I had to take special considerations during the pixel-level collision check. I couldn't figure out why I was sometimes getting an exception during collisions. I realized that the animation frame could change as the pixel-level check was iterating through pixels. To solve the problem, I simply grabbed a reference to the appropriate frame as soon as I entered the pixel-level checking method; rather than dereferencing the animated sprite for its surface each time I needed it within the loop.

5.4.2 Stop the Bouncing! (top)

Each instance of SolidEntity maintains a list of other SolidEntity objects it is currently colliding with. This is useful because I do after-the-fact collision detection, which means the objects are already overlapping when I detect the collision. A collision triggers a reaction that varies depending on the types of objects involved (i.e., planet, ship, bullet). However, the reaction typically involves an change of Velocity (and usually direction). If the resulting bounce is unable to separate the two objects in the next Tick handler's Update call, then the collision will be detected again (and reacted to again). As you might imagine, this can result in an amusing, rapid-fire, ping-pong effect. However, this is not particularly desirable behavior for my space ships. Thus, the list is maintained so that: if a collision is detected more than one, the reaction will be bypassed. To keep it current, the list is pruned each tick when the collision checks happen.

5.4.3 Ships Being Damaged by Their Own Bullets (a.k.a., Et tu Brute?) (top)

In the course of testing the near-final versions of the game, I came across a gameplay bug that occurred often enough to be detrimental to the overall experience. A player could fire a shot shortly before dying, respawn, and get damaged by their own bouncing bullet! This was due to the fact that each Bullet class was linked via reference to its creating ship. If the ship was destroyed, then the player would receive a new ship, which the bullet wouldn't recognize. It was a simple fix: I simply associated each bullet with its Player rather than its Ship. This was

really the point in the program's evolution that I decided to make a Player class; it was an idea I had considered, but this was the "last straw."

5.4.4 Max-Bullets Exploit (a.k.a., Suicide-to-Reload Trick) (top)

Another issue I discovered while testing was more of an exploit than a bug. The count of live bullets is tallied at the cannon level, and bullets persist even after a player's death (I feel this adds a fun element). So, a player could have more than the maximum number of live bullets by firing off a rapid barrage, quickly dying, and then firing more. To fix this, I could simply move the counting of live bullets from the Cannon class to the Player class. However, it's hardly a serious issue, and I just didn't get around to fixing it. However, I did add it to my "Future Work" list (below).

5.4.5 Framerate Independent Movement and Predictive Collision Detection. (top)

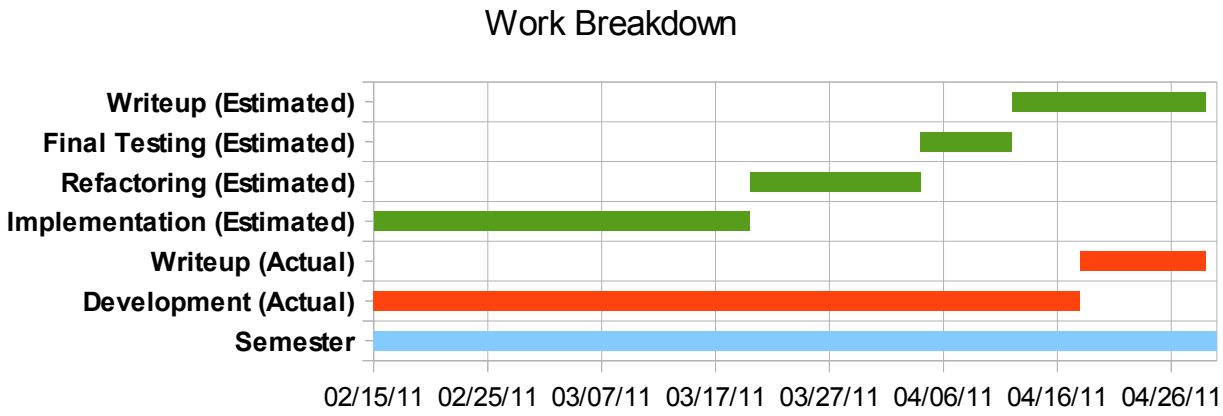
Ideally, I would add support for framerate independent movement, which moves a particle according to the amount of time that has passed since the Tick handler was last called; rather than a static distance per Tick. It is possible to do, but SDL.NET clearly wasn't designed with framerate independent movement in mind. Implementing it would mean discarding SDL.NET's ParticleSystem almost entirely: since theirs is designed to progress only in non-fractional units. Additionally, framerate independent movement would necessitate some form of predictive collision detection. If a SolidEntity can move more than one "unit" in a Tick, we need to know if it will cross-paths (i.e., collide) with another object; otherwise we could move one object through another. Predictive detection would allow the program to notice and react to those collisions. In the end, I decided that the extra effort wasn't worth it. I had made a decision at the start to use SDL.NET, so I felt it was right to use it as-intended.

5.4.6 Optimization (top)

As I neared the end of the development phase I came to a shocking realization. The game worked very smoothly on many machines, but on my slower ones the game sometimes became unplayably slow! I used a performance profiler to hone in on the slowdown points. As it turns out, the performance hit came from the fact that I had not converted the images I loaded from the disk into a uniform pixel format. So, the conversions were being done each time, on-the-fly! Once I made those simple changes, the speed of the game greatly increased. Even my lowly netbook can now maintain the game's set framerate of 30fps.

5.5 Work Breakdown (top)

I initially broke the development of the project into distinct phases of coding, refactoring, and testing. I carefully separated each task, and assigned a specific schedule to it. However, given the nature of evolutionary programming, and the fact that I served as my own client, I ended up adhering to a more informal system of short development cycles (implementing, testing, and refactoring). The project still reached completion at the planned time, and there were no code-like-hell sessions involved. The chart below is simplified from the chart in my original proposal.



6. Results

(top)

I'm quite satisfied with the results as a whole; both as the client and the developer! The game is very playable and fun. However, it would be nice to have a computer player to compete with, since computer scientists often don't have real friends.

Check out the screen shots of the finished product below.

6.1 Final Program

(top)

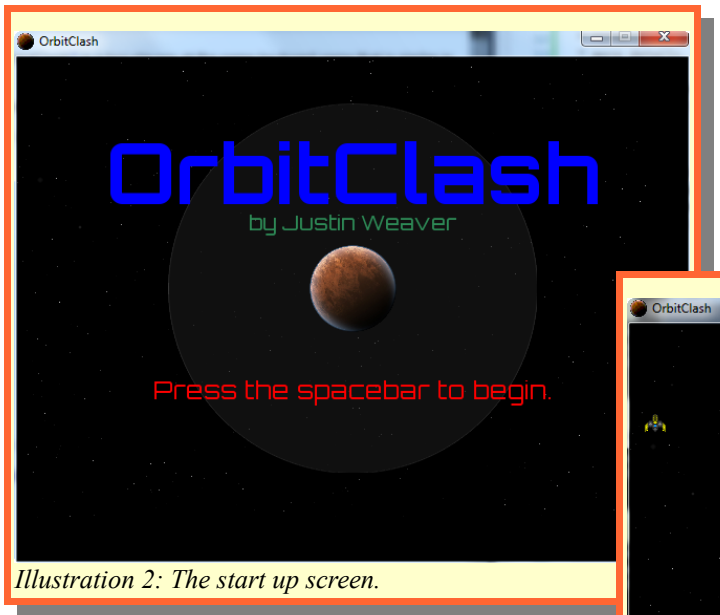


Illustration 2: The start up screen.

Illustration 3: The ships in their initial positions.

Illustration 4: An epic space battle!

6.2 Performance

(top)

SDL.NET uses Microsoft .NET Interop services to call the unmanaged SDL libraries, which incurs some overhead. In spite of this, the game does not put a significant strain on the system. In fact, its exact requirements are unknown, but low enough that the maintainable frame rate isn't likely to be much of an issue. However, it would still be more correct to implement framerate independent movement and predictive collision detection.

6.3 Public Release

(top)

I published the game on GoogleCode. The site contains a source code (SVN) repository, as well as binary and source code archive downloads. The game's homepage http://orbitclash.googlecode.com.[Illustration 5]

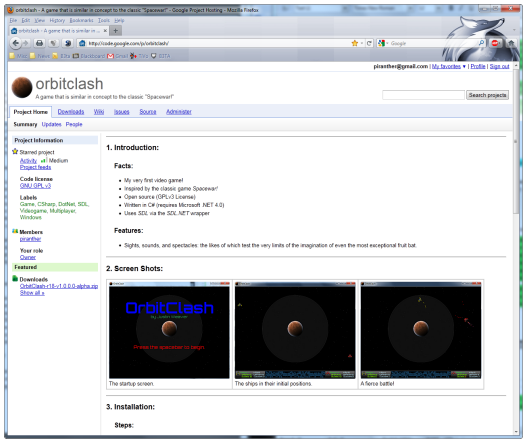


Illustration 5: The OrbitClash homepage.

6.3.1 User Manual

(top)

Operating the game is exceedingly simple. However, the OrbitClash user manual is included on the game's home page and in the ReadMe.txt file bundled with each distribution. I made a tactical decision not to duplicate it again in this document.

6.3.2 Licensing Details

(top)

I chose the GNU General Public License[R5] (GPL) for OrbitClash, because it is compatible with all of the licenses of the other resources used in this project, and because it prevents my code from being used in some random closed-source, commercial product (theoretically).

SDL and SDL.NET are both licensed under the GNU Lesser General Public License (LGPL).[R6] Which is an open source GNU license commonly used with libraries.

The two ship graphics were created by "JVI i I{ I{," who made them freely available[R7] under the conditions that they be attributed for their work, and that they be notified (via thread post) when their work is used; I did both.

The planet graphic was created by Christian Hollingsworth[R8] and licensed under the Creative Commons Attribution 3.0[R9] license.

The sound effects I used were all obtained from SoundBible.[R10] Most of the sounds were created by Mike Koenig, and released under the Creative Commons Attribution 3.0 license. For a comprehensive list, see the "Sound Attribution.txt" file in the Sounds directory of either the binary or source code distributions or OrbitClash. The ship thruster sound was created by dobroride, and released under the Creative Common Sampling Plus 1.0[R11] license. The ship warping-in sound was created by snottyboy, and released under the Creative Commons Attribution 3.0 license.

The only font I used is called Orbitron, and it was created by Matt McInerney,[R12] and released under the Open Font License.[R13]

I took special care to provide complete and specific licensing details in every distribution of the binary and source of the game, so as to be sure not to violate any of the sub-licenses.

6.4 Wish List

(top)

- Tally "live" bullet count in the Player class, instead of the Cannon class, to prevent max-bullet exploit.
- Graphic-effects when bullet hits ship
- Graphic-effects when ship hit ship
- Graphic-effects of ship damage level
- Configuration
- Joystick support
- Framerate-independent movement & predictive collision detection
- Ability to pause/unpause the game
- "Press 'H' for instructions" on main title screen
- Give SolidEntity objects "mass" to make physics more realistic
- Support higher resolutions

- Network play
- Power-ups
- Player profiles
- Choice of ships /w various specs
- AI to play against
- Installer for binary distribution

7. Summary

(top)

For this project, I served as my own client, so I found it sensible to use an evolutionary prototyping design methodology. My goal was to develop a video game inspired by the classic game *Spacewar!*

The result is a very playable, open-source game. I (the client) have declared the project a brilliant success! As mentioned above, to download the finished result, visit http://orbitclash.googlecode.com.

I learned a little bit about some very basic game programing (graphics, physics, sound, and player input) along the way. I also learned that collision detection is not as simple as it seems. I feel like my work was a worthy introduction to the basic concepts of video game design.

8. References

(top)

- [R1] Wikipedia contributors. "Spacewar!." Wikipedia, The Free Encyclopedia, 18 Jan. 2011. <http://en.wikipedia.org/wiki/Spacewar!>. Web. 24 Jan. 2011.
- [R2] Microsoft. "Microsoft .NET Framework." Web. Accessed 01 Feb 2011. <http://www.microsoft.com/net/>.
- [R3] SDL. "Simple Directmedia Layer." Web. Accessed 01 Feb 2011. <http://www.libsdl.org>.
- [R4] C# SDL. "C# SDL - Main Page." Web. Accessed 01 Feb 2011. <http://cs-sdl.sourceforge.net/>.
- [R5] GNU. "GNU General Public License (GPL)." Web. Accessed 18 Apr 2011. <http://www.gnu.org/licenses/gpl.html>.
- [R6] GNU. "GNU Lesser General Public License (LGPL)." Web. Accessed 18 Apr 2011. <http://www.gnu.org/licenses/lgpl.html>.
- [R7] JVI i I{ I{. Ship Graphics. Web. Accessed 18 Apr 2011. <http://gmc.yoyogames.com/index.php?showtopic=159419>.
- [R8] Christian Hollingsworth. Planet Graphic. Web. Accessed 18 Apr 2011. <http://www.flickr.com/photos/smartboydesigns/3797823072/>.
- [R9] Creative Commons Attribution 3.0 License. Web. Accessed 18 Apr 2011. <http://creativecommons.org/licenses/by/3.0/>.
- [R10] SoundBible. Web. Accessed 18 Apr 2011. <http://soundbible.com>.
- [R11] Creative Commons Sampling Plus 1.0 License. Web. Accessed 18 Apr 2011.

<http://creativecommons.org/licenses/sampling+/1.0/>.

- [R12] Matt McInerney. "Orbitron Font." Web. Accessed 26 Apr 2011. <http://theleagueofmoveabletype.com/fonts/12-orbitron>.
- [R13] Nicolas Spalinger & Victor Gaultney. "SIL Open Font License 1.1." Web. Accessed 18 Apr 2011. <http://scripts.sil.org/OFL>.

8.1 Useful Tools

(top)

- **Audacity** - *The Free, Cross-Platform Sound Editor.* <http://audacity.sourceforge.net/>.
- **The Gimp** - *The GNU Image Manipulation Program*. <http://www.gimp.org/>.
- **Tortoise SVN** - *A Subversion client, implemented as a windows shell extension*. <http://tortoisesvn.tigris.org/>.