Let it all hang out

Alex Attar - [redacted]

David Campbell - [redacted]

Jeffrey How - [redacted]

Addison Rodomista - [redacted]

April 19, 2013

This page intentionally left *naked*.

Table of Contents

Executive Summary

Superflash is a 2D action arcade game in which the player controls a streaker whose goal is to attract attention and wreak havoc at a variety of public venues. The streaker must dance, dodge and disgust in order to complete objectives, win notoriety points. With its stylized visuals, catchy music, action-packed game play and overall ridiculousness, players of all ages will want to “let it all hang out”.

Overview

Superflash is a 2D side-scrolling action arcade game for PC, aimed at players 12 and up who enjoy action-packed and fast-paced games that are as funny as they are fun to play. Playing like a cross between *Pacman* and *Infectionator*, the player controls a curly-haired teen who is anxious to draw attention to himself by engaging in the original American pastime: streaking in crowded areas. This pastime is not for the faint of heart however, and to succeed the player will have to draw attention to himself while avoiding security officials who don't enjoy being flashed on the job. Taking advantage of only his surroundings, his wit and his dance moves, the player will have to find ways to gain as many notoriety points as possible before his inevitable capture.

Avoiding the security officials is key to a successful streak, and the player will have to engage in some strategic maneuvering to stay free from the clutches of law enforcement officials. While the run-of-the-mill police officers will mindlessly seek the player, the riot police officers and Robo-Cops will instead conspire strategically to trap the player. Meanwhile, innocent bystanders will do whatever they can to flee the scene of the streak.

Best of all, notoriety points are not difficult to obtain, but instead can be earned by doing almost anything at all, although more notoriety points will be awarded for more risky behaviour. Aided by celebratory animations and sounds, the players will gain the immense satisfaction of scoring lots and lots of notoriety points. Notoriety points reflect the overall havoc that has been created by the streaker. As such, dancing, escaping security guards and running pedestrians over will gain notoriety points. At the most basic level, notoriety points are essential to achieve new high scores.

Together, these mechanics make *Superflash* a very entertaining and enjoyable game. There are numerous fast-paced internet games which like *Superflash* are based on silly concepts and over-the-top scoring. Similarly, there are many games that are based on dodging enemies and staying alive as long as possible. However, *Superflash* is more than about getting a high score or running away from enemies. Unlike these other games such as *Miami Shark*, the enemies in *Superflash* pose a legitimate threat to the player, and unlike games like *Pacman*, the player will have to do more than simply avoid enemies to do well in the game. While other games offer one of *Superflash's* elements to the player, the unique combination staying alive and playing creatively to achieve a high score is unique to *Superflash*. As such, the game has the same initial appeal as a humorous online game, but has the lasting appeal and replay value of a larger game.

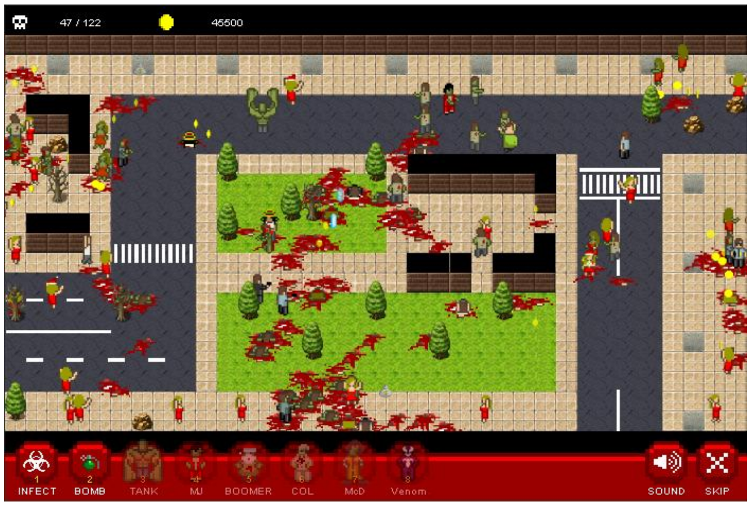
Related Games

The simple and unique combination of game mechanics is not the only appeal of *Superflash*; the comically-stylized art is reminiscent of *Castle Crashers* while the upbeat and humorous music will remind players of the mayhem that came along with playing classic party games like *Mario Party* and *Mario Kart*. While the focus of the game is to be fun, challenging and contain well-programmed AI, *Superflash* is also an engrossing game as a result of its art assets and overall aesthetic. High scoring and fun gameplay are great, but the eye-catching art and interesting sound design will help truly engross the player.

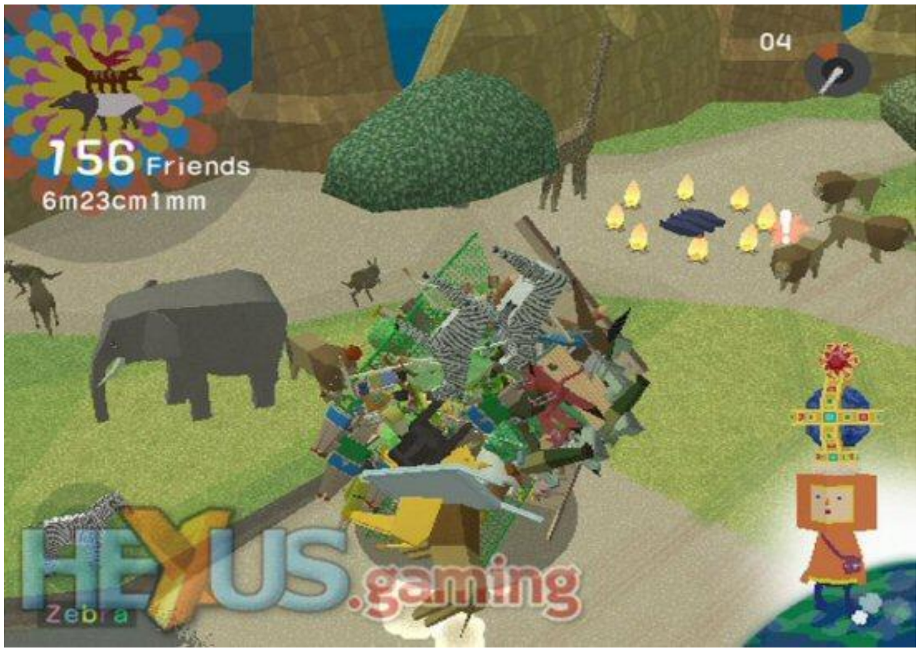
While the art style and music are reminiscent of *Castle Crashers*, *Mario Kart* and *Mario Party*, the gameplay instead contains elements from such games as *Pacman*, *Infectionator*, and *We Love Katamari*.

In *Pacman*, the player must guide Pacman, a yellow pie-shaped character, around a maze so that he can eat all of the pellets placed in the level. The presence of four ghosts in the maze makes this task more difficult, as the ghosts kill the player on contact. Pacman's only defenses against these ghost are the four power pellets scattered in each stage which render Pacman invulnerable to the ghosts for a short period of time. Like Pacman, our game will take advantage of enemy NPCs following different AI strategies, leading to a very challenging and seemingly complex result. Our game will also try to incorporate the frenzied and over-the-top feel of *Pac-Man Championship Edition DX.* However, unlike *Pacman* our game will focus on obtaining high scores by performing a variety of different tasks and interacting with NPCs in a variety of ways, unlike Pac Man in which the player can only gain points by eating pellets, power-ups and NPCs.

In *Infectionator*, the player attempts to obtain the greatest number of casualties by spreading a zombie virus worldwide. By selecting a location from which to release the virus, the player turns NPCs within a certain area into zombies, which in turn infect others around them. The larger and more populous the environment, the more chaos ensues. We plan to have a similar fast-paced and frantic feeling to our gameplay. Some of the enemy AI will incorporate similar group cohesion and fleeing ideas. Unlike *Infectionator*, our game will have slightly more player control, as it will not be merely a "point-and-click" game.



In *We Love Katamari,* the player pushes around a small sticky ball onto which objects of lesser size stick. The player's goal is to stick objects to the ball until it reaches the size required to complete the level. As the ball grows in size, the player is able to stick larger objects to it, eventually picking up entire buildings. While the core mechanic of our game will be different, we want to capture the same sense of cartoonish chaos and general ridiculousness of *We Love Katamar*i.



Overall, *Superflash* will seek to captivate the players of all ages with its addictive fast-paced action, its challenging AI, its visually and sonically appealing design, and overall ridiculousness. While the game bears some similarities to other puzzle games such as *Pacman*, *Infectionator* and *We Love Katamari* among others, the original combination fast-paced action, replay value and good level design will ensure that *Superflash* constitutes a unique and worthwhile offering for fans of both the simple survival game and high-score style arcade game.

Game Characters

The Streaker

The Streaker has lived a long and fulfilling life of tomfoolery and mayhem. Raised by Nudist parents, the Streaker spends his days exposing himself to the general public in an effort to cause mass disruption. The Streaker, while almost always nude, will only ever be seen wearing clothes while showering in public gym showers, where he prefers to wear a blue bathing suit for no reason except irony.

The Streaker is the main character of the game. There are animations for standing, running, falling, Superflashing, and dancing.

The Dumb Cop

After years of being a bully in High School, the dumb cop was left uneasy in the real world with everyone else maturing and moving on; leaving no one for him to pick on. He was able to find a suitable career as a Cop in the police force. Now he spends his days hassling jaywalkers and ticketing teenagers drinking alcoholic beverages in parks.

The dumb cops are the first enemy the player will encounter in the game. There are animations for standing, running, attacking, and falling.

The Smart Cop

Having been rigorously trained in the Special Forces, the smart cops have a thirst for stopping crime. They have been taught to suppress all possible emotions, and have strict concentration on the task at hand.

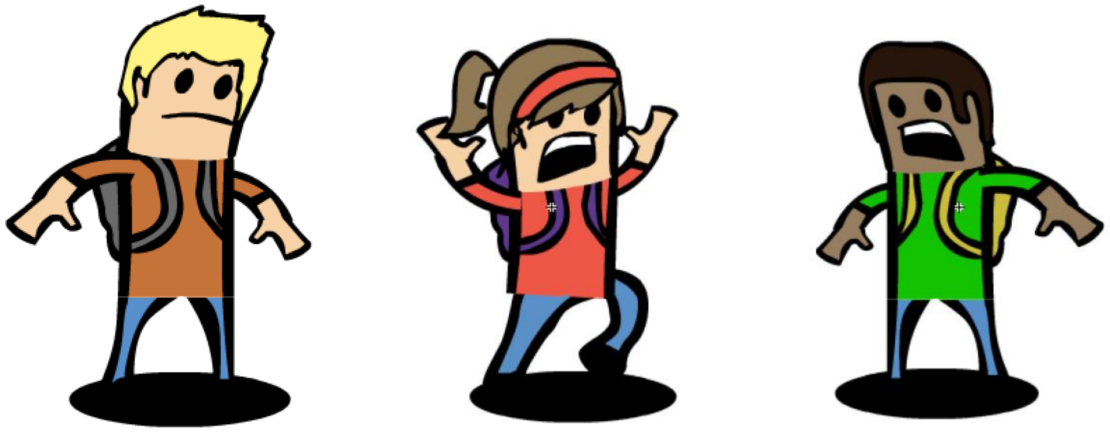
The smart cops will spawn over time intervals throughout the game. They contain the same animation variations as the Dumb Cops.

The Robo-Cop

Created by the infamously conservative mad scientist, Dr. Krubapulstein, the Robo-Cop was designed for one reason: terminate all nudity. Once the Robo-Cop has been exposed to any form of nudity, it will stop at nothing to track down and eliminate the perpetrator.

The Robo-Cop contains only animations for standing, running, and attacking. The Robo-Cop cannot be knocked over.

The Students

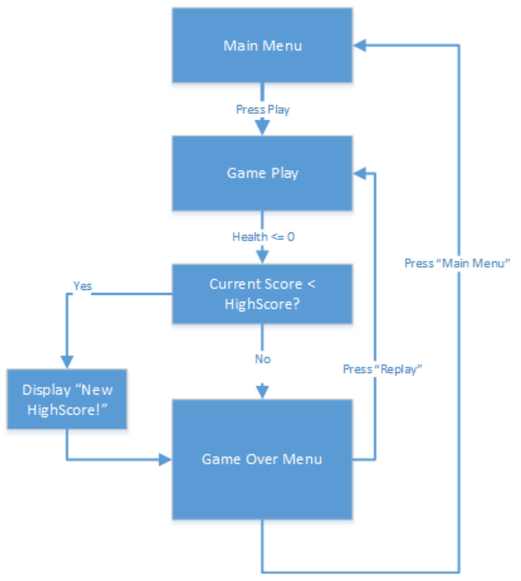


What drives the students is the same thing that drives all teenagers. To learn, live, and have fun. Having grown up in a small town, they have been exposed to very little of what the world has to offer, and every day they are learning more about what's around them, and more importantly, themselves.

Student NPCs are scattered all around the level. They contain animations for standing, walking, fleeing, falling, and getting up.

User Interface Storyboards

UI State Machine

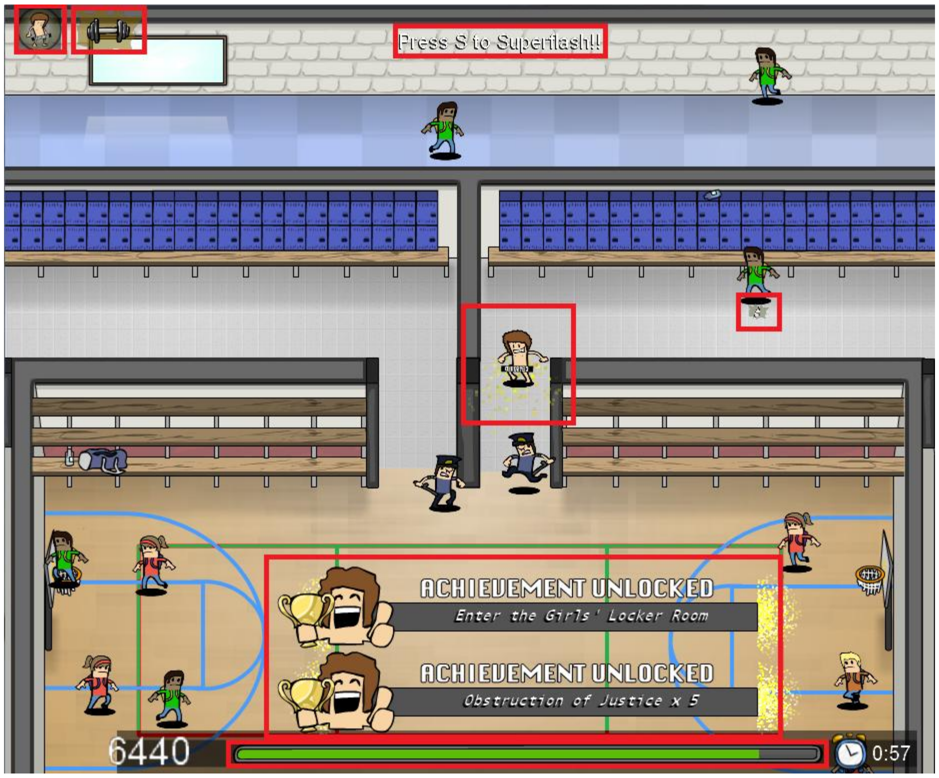


Main Menu

SUPERFLASH
CENSORED
PLAY

From the Main Menu, the player must select the "Play" button to navigate to the core game play.

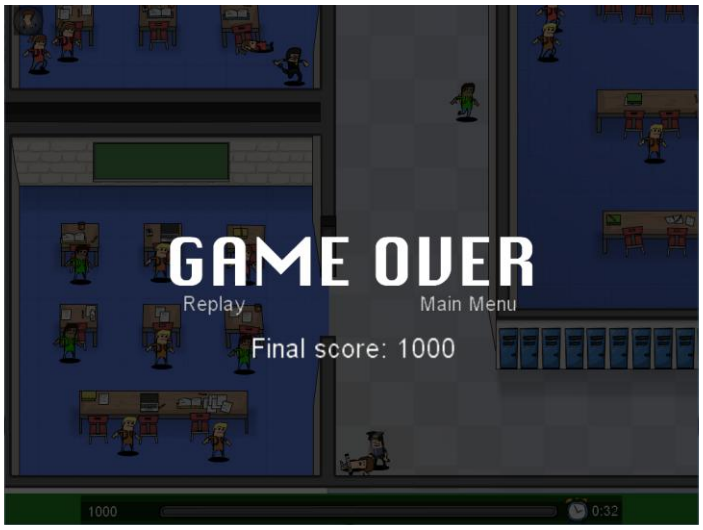
Game Play



During game play, there are a variety of components on the screen detailing the state of the current game. On the upper left of the HUD, there is an icon detailing whether or not the Superflash has fully recharged. By default this will be transparent, but will flash once the Superflash is ready. If and when the user has acquired a power-up (like the one shown in the image above), it will appear as a small icon immediately to the right of the Superflash icon.

Across the bottom of the screen, the HUD displays the current score, the player's current health, as well as the total time elapsed since the game has commenced. When a player unlocks an achievement, a banner will pop up directly over the HUD providing the player with the name of what they have accomplished.

Game Over Screen



Once the player's health drops to 0, a game over screen will be presented to the player. If their current score is greater than their high score achieved in a previous play, they will be alerted with a "New Highscore!" message. From this menu the player can choose to navigate to either the Main Menu, or start a new game right away using the Replay button.

Technology Plan

Programming

All programming for this project was done by the four members of this team. No external source code or libraries were used for this game.

When coding some of the artificial intelligence behaviors and physics simulations for our game, concepts and techniques were taken from material covered in the slides, which originally came from Ian Millington's *Game Physics Engine Development* and Ian Millington and John Funge's *Artificial Intelligence for Games* (*2nd Edition)*. Other concepts such as collision optimization and dynamic A* cost updating were either our own ideas or inspired by class material and implemented by us. The following software was used for programming:

Microsoft Visual Studio 2010 Professional (C#)

Microsoft XNA Game Studio 4.0

Documentation

Documentation was done by all members of the team. The following software was used to create the documentation:

Google Docs

Microsoft Office

Source Repository

For giving all members of the team access to a shared source code, an Apache Subversion (SVN) repository was created. The following software was used to work with the repository:

Google Code

Tortoise SVN

Graphics and Art Assets

All art assets including characters and animations, the game level, the menus, and the in game objects were created from scratch by Addison Rodomista. The following software was used for creating art:

Adobe Flash CS6

Adobe Photoshop CS6

Sound

All sound in the game including vocal and environmental sound effects as well as music was created by David Campbell. All vocal exclamations were recorded from scratch, while power-up and environmental sounds were created by modifying and combining sounds from a free to use sound library. Unfortunately, this sound library was obtained via a Concordia music professor and as such the original name and location of the library is not known. For convenience, the sound library has been uploaded on David Campbell's Google Drive[1]. The following software was used to create the music and sound:

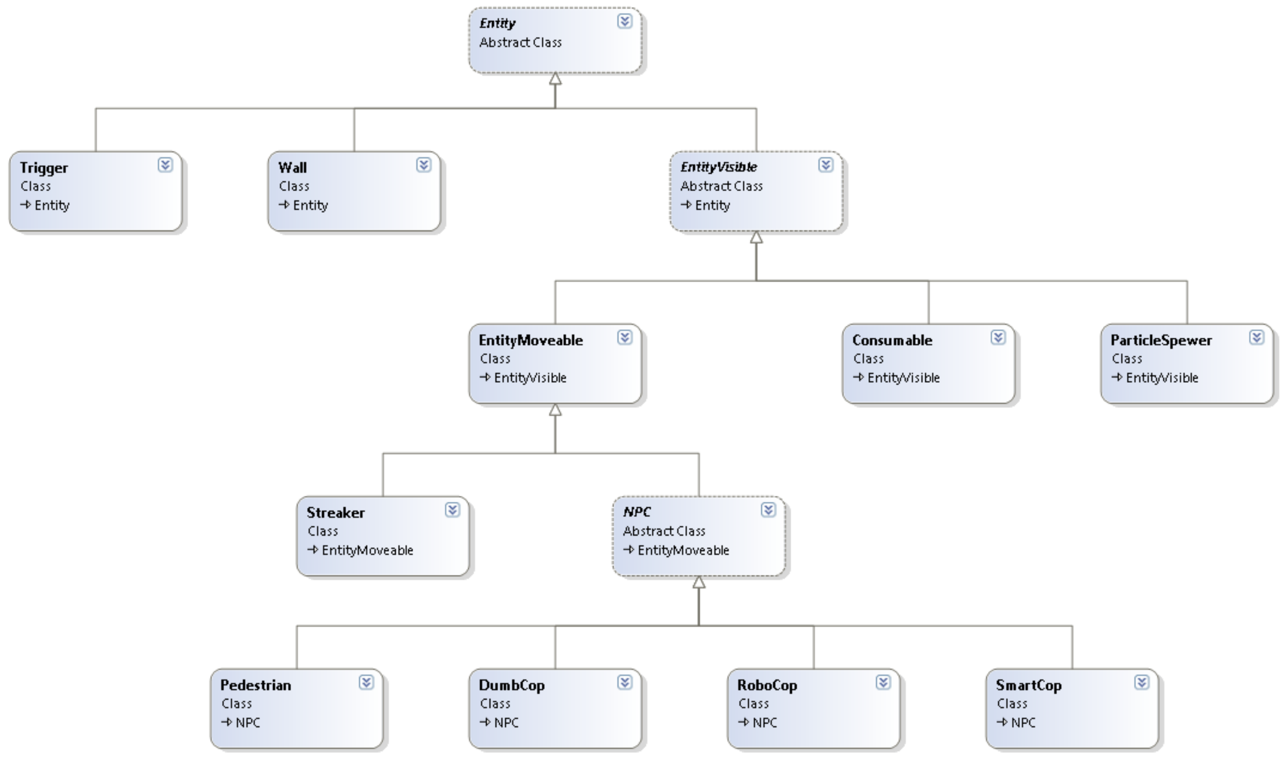
Guitar Pro 6

Reaper

[1] https://docs.google.com/file/d/0B_laCCXfxLFtdnhhcXRSRTQyZ0k/edit?usp=drive_web

Software Architecture

Entity/Component



Entity

A key programming decision we made was to use the Entity/Component game design pattern[2].

Alone, an Entity is just an empty container with a position and bounding box. But as seen above, essentially any interactive object in the game world inherits from Entity and its subclasses. Rather than encapsulating all the data about an entity in the "Entity" class, the data is encapsulated in separate, domain-specific components which are added to the container

[2] Inspired by http://gameprogrammingpatterns.com/component.html

depending on which subclass is being implemented. These components (DrawComponent, PhysicsComponent2D, and MovementAIComponent2D) will be discussed below

Components

PhysicsComponent2D

The PhysicsComponent2D encapsulates data and methods related to an entity's positional and physical state. Attributes include values for velocity, acceleration, direction, mass and more. The functions in this class are used to update and manipulate such data and perform tasks such as collision resolution.

MovementAIComponent2D

The MovementAIComponent2D includes all data and functions related to rudimentary AI movement. Using information such as satisfaction radii as well as target values for position, velocity, orientation and rotation, this class can implement movement functions such as seek, wander, pursue, flee, align, etc. Each of these movement functions follow the steps outlined in the class slides.

DrawComponent

The DrawComponent class encapsulates all tasks related to drawing the sprites and animations of its associated entity. It includes drawing related values such as scale, color, depth, alpha, etc. Aside from its Draw and Update functions, it also include functions that Stop, Reset and Play animations. DrawOscillate is a subclass of DrawComponent, which adds the ability to oscillate in alpha value and/or scale.

Entity Subclasses

Trigger

Triggers are essentially invisible rectangles which activate events. In our case, such events include winning achievements for entering particular rooms. Therefore, when the player walks in a room that has an associated trigger entity, the related achievement is “triggered”.

Wall

Walls are impassable rectangles which may or may not be see-through. As such, obstacles such as tables are see-through, whereas walls are not, allowing for more complex decision making to be used.

EntityVisible

This subclass of Entity adds a DrawComponent into the container. Some concrete subclasses of EntityVisible include Consumable and ParticleSpewer. A Consumable is an object than can be picked up by the player for a power up.

EntityMoveable

EntityMoveable is a subclass to EntityVisible. It adds a PhysicsComponent2D to the entity. In contrast to a Consumable, which stays in one place, an object inheriting from EntityMoveable has the ability to move around and change its orientation. An example of a concrete EntityMoveable is the Streaker, who can run around the level depending on player input.

NPC

NPC is a subclass to EntityMoveable. It adds a MovementAIComponent2D to the entity. Essentially, this abstract class allows us to create characters who can think and make decisions on their own. Concrete child classes include Pedestrian, DumbCop, SmartCop and RoboCop. The concrete classes include code related to higher level decision making for each character. Please refer to the Artificial Intelligence section for more information.

Game State Related

World

The world holds all information relating to the current game state. It is composed of what map is being used and what entities are on the map, such as the streaker, cops, consumables and more. It is mainly takes care of calling the update and draw functions for all entities in the game state. The world is also used to detect collisions between entities in the world. After a detection it delegates the resolution task to the entities who handle the collisions themselves

Map

The Map class uses parsing to read Map Editor output (text file) and uses the data to create and instantiate entities, nodes and triggers according to positional data in the text file.

HUD

The HUD contains both DrawableComponents and FontComponents, and displays all the UI elements to the player during game play. Such information includes the Streaker's health, current power-up, score and the elapsed time. The HUD also handles the interpolation of animated visual components such as the health bar decreasing, and the score flashing as it increases. Lastly, the HUD also takes care of fading to the game over screen and displaying post-game related information.

Camera

The Camera class contains the data required to set the view of the game. It has a position, representing the center of the view, and dimensions which are set to those of the viewport. It also includes an optional scale value allowing the view to be zoomed in and out. The camera's attributes are used by the game to create a transformation matrix for the main SpriteBatch used by the XNA framework to draw all sprites. Therefore, unless specifically accounted for, all sprites are affected by the camera's position, allowing for a "world" larger than the viewport dimensions.

Managers

All managers were implemented using the Singleton design pattern. This is because they implement functionality that is needed only once in the entire game and is likely needed by the entire game. The Singleton design pattern makes their functionality easily accessible without unnecessary duplication.

Achievement Manager

The Achievement Manager handles the updating and drawing of all achievements in the game. When the Achievement Manager is updated, it checks each if achievement in its list is achieved (individual achievements determine the criteria for this) and creates pop-up “toasts” and assigns points if they are. When the Achievement Manager is drawn it draws all existing “toasts”.

Data Manager

The Data Manager encapsulates all of the game’s statistical data. This includes the player’s health and score, as well as miscellanea such as the number of Superflashes used and the number of pedestrians knocked over. These values are public and since the class implements the Singleton design pattern, they are effectively global. As such other classes, such as the HUD or achievements, can check any criteria they need at any point. The Data Manager is also responsible for creating and displaying the small score number pop-ups that occur at certain events.

Input Manager

The input manager handles all keyboard key-mappings and gamepad button-mappings. It also detects when an Xbox controller is connected to the system. If so, then gamepad controls override keyboard controls.

Sound Manager

This class handles any sound related tasks. It includes a dictionary and functions, which can be used to retrieve all sounds and play them. This class also adds randomness to playback and pitch. With respects to playback, randomness was added to handle the issue of too many sounds being played all the time. As for pitch, randomness was used to mitigate monotony and give variety to the sounds.

Sprite Database

The Sprite Database class is responsible for loading all of the game's sprites and contains all of the game's animations. The loadSprites method is called once per execution of the game at the start, after which all sprites are loaded and accessible by the rest of the game. The Sprite Database was moved outside of the core game and into a small StreakerLibrary in order to facilitate sharing assets between the game and the level editor.

Artificial Intelligence, Collision and Advanced Features

AStar

The AStar class is a static class whose sole purpose is to calculate a path between two positions using a set of nodes and edges. It encapsulates no data, only this functionality.

Flock

The Flock class represents a flock of NPCs. It encapsulates the functionality for calculating the cohesion, separation and alignment vectors for any member of the flock and which NPCs are members of that flock. It also contains the weights to be used by those vectors.

QuadTree

The QuadTree class is an implementation of a quadtree for Walls. It is a standard implementation of a quadtree and is able to store Walls or retrieve a list of Walls that are relevant to either a given point or a bounding box.

Node

The Node class is used for path-finding and is effectively half of the requirements for a path. It contains its own position, a list of its associated edges and whether or not it is a “key” node. Nodes are responsible for updating their connected edges.

Edge

The Edge class is used for path-finding and is the other half of the requirements for a path. It keeps track of the two connected nodes, its cost and its weight. Edges will erode their weights on update.

Line Segment

The Line Segment represents a mathematical line with two endpoints. Given a start and an end, the class computes the general equation of the line between them[3] and contains useful methods such as line-line intersection, line-box intersection, and retrieving the X or Y coordinate of a point on the line given the other coordinate.

Particle Spewer

The Particle Spewer is a Drawable Entity used for emitting particles. The Particle Spewer is constructed with all necessary data (position, particle and emitter count, color, speed and angle). The Particle Spewer can then be started and stopped, automatically creating and updating its particles. Particles are created randomly within the ranges designated by the Particle Spewer.

[3] $Ax + By + C = 0$

Achievement-Related

Achievement

The Achievement base class contains the minimum requirements for any achievement. These are a name, description, point value, an Update method and an IsAchieved method. All other achievements inherit from this class.

Achievement Toast

The Achievement Toast class is a part of the UI and represents the banner displayed when an achievement is unlocked. They are called “toasts” because they are designed to pop up from the bottom of the screen like toast from a toaster. They are instantiated with a related achievement from which they take the name and a lifespan which determines how long they are visible.

Controls

Keyboard Controls

Up Arrow: Move the player Up

Right Arrow: Move the player Right

Left Arrow: Move the player Left

Down Arrow: Move the player Down

‘S’ Key: Perform a “Superflash”

‘D’ Key: Perform a Dance

Spacebar: Earn an achievement

Xbox Controller

Left Analog Stick / D-pad: Move the Player

‘X’ Key: Perform a “Superflash”

‘A’ Key: Perform a Dance

Level Editor

‘M’ Key: Cycle through editing mode

‘W’ Key: In some modes, change the type of object being placed

‘1’, ‘2’, ‘3’, ‘4’ Keys: Change the type of NPC or consumable being placed

‘S’ Key: Save the current objects to “out.txt” in the same directory as the level editor

‘L’ Key: Load the “level.txt” file from the StreakerLibrary folder

Left Mouse Button: Place the object associated with the current editing mode

Right Mouse Button: Remove the clicked object from the map

Left Click and Drag (box or trigger mode): Create a box. Release to place

Left Click and Drag (node mode): Create edges between nodes. Release to set.

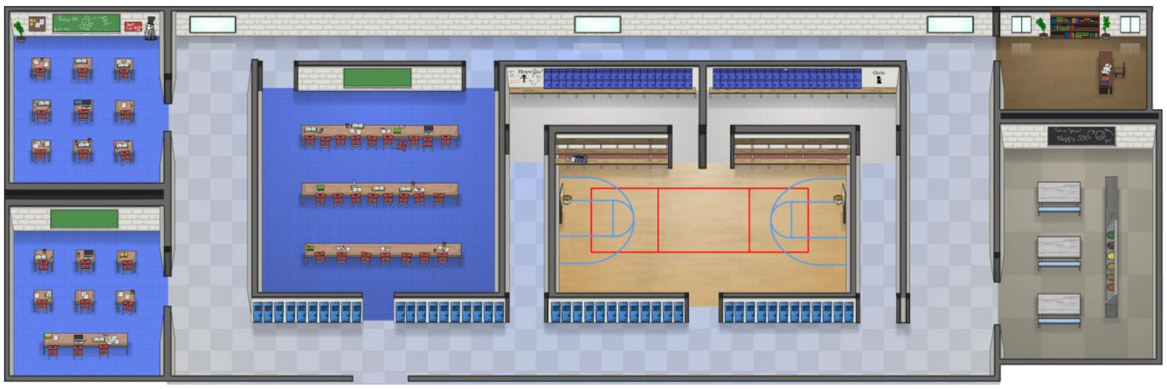
Level Design

Background

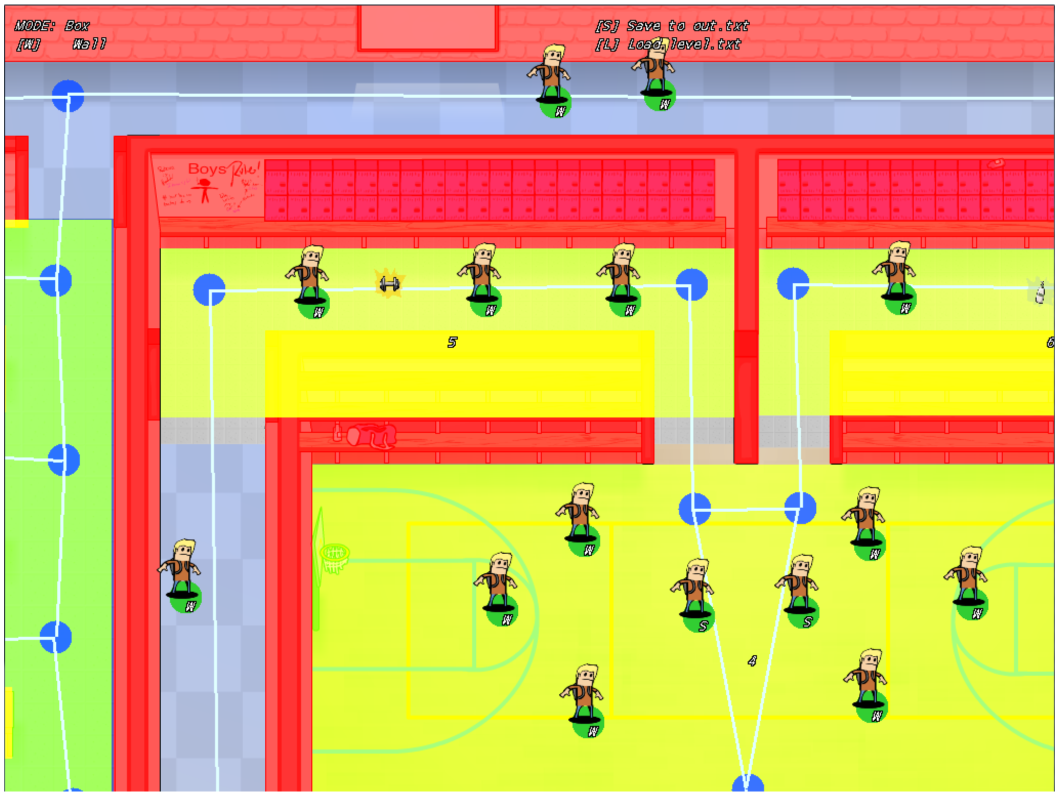
The level was designed with a large, well connected center and segregated rooms on the sides. While the player is in the hallways, the lecture hall or the gymnasium, they almost always have several options for paths to take, as do the NPCs. In the rooms on the edge, there is only one way in or out, making them dangerous. However, these rooms have many pedestrians which allow the player to accumulate many points and the principal's office in the top right is where the energy drink, the most valuable power-up, appears. The player must therefore weigh the danger versus the rewards.

The early design of the map was much more vertically focused. Originally, the gymnasium was located above the lecture hall, giving the level a square shape. Given the nature of the animations (the characters do not have animations to run upwards or downwards, only left or right) and the perspective of the art, the level was modified to be more horizontally focused.

Overview of Map



Level Editor



Aside from the images, the level was primarily built using a custom level editor. The editor allows the user to place all available game objects into the map, then save it to a text file that can be read by the game.

The first assets which can be placed are walls. Using the mouse to click and drag red rectangles onto the image, the user can quickly fill the map with see-through walls (bounding boxes which the character can see through but not move through, such as tables and chairs) and solid walls.

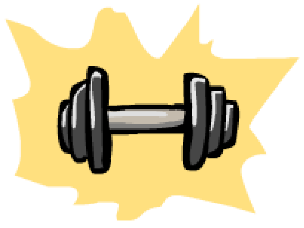
Nodes can also be added to the level with this map editor tool. By clicking on the image in node mode, a path finding node can be placed in the level. This node can either be a regular node or a key node to which smart cops will head when they have lost the streaker in order to regain sight of him. Key nodes are placed outside of rooms in the hallway, in order to increase the odds of catching sight of the streaker again. Nodes can then be connected to create a graph of the level.

All NPCs in the game can also be placed in the map. Whether a pedestrian, a dumb cop, a smart cop or a Robo-Cop, the NPC can be given an initial location simply by clicking on the map. The same idea can be used to place consumable item spawn points on the map, as well as the streaker’s starting location.

Mechanics Analysis

Power Ups

Several power-ups are spawned throughout the level over a period of time. Once acquired, the player will lose the ability after 10 seconds.



Steroids

Gives the player the ability to knock down both Smart Cops and Dumb Cops. The player is awarded points for each cop knocked over.

Energy Drink

Gives the player a significant boost in speed.

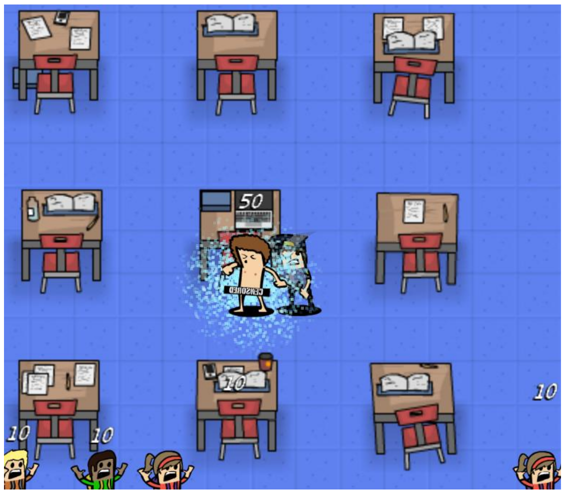
Nike Jordans

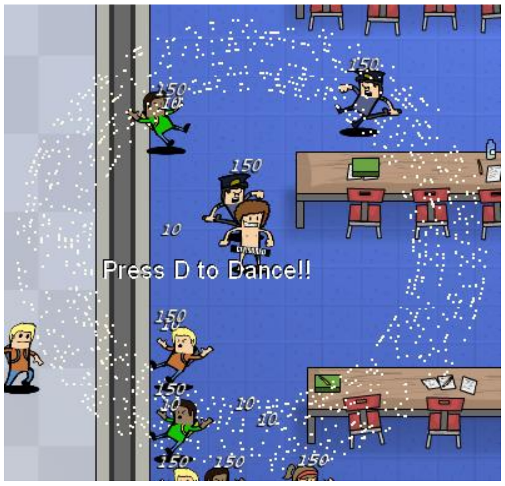
Increases the player's acceleration, allowing for him to take much sharper turns.

Aveeno Moisturizer

Makes the player extra slippery and invulnerable to attacks.

Player Actions and Interactions





Dancing

By pressing the “D” key on the keyboard, or the ‘A’ button using a controller, the player will initiate a dancing mechanic to taunt enemy Cops and gain points. The image on the left is a picture of the Streaker dancing.

Superflash

By pressing the ‘S’ key on a keyboard, or the ‘X’ button on the controller, the player will perform a super flash that knocks down any NPCs within a specified radius of the player. This can help the player get out of areas with a high concentration of Enemy NPCs and also be a way to load up on points. The image on the right is a picture of the Streaker executing a Superflash.

Achievements

Achievements are an integral part of what makes Superflash so fun, and can be unlocked in the following ways:

1. **Easiest Achievement Ever (1000 points):** Exist for 2 seconds
2. **Long Streak (1000 points):** Streak for 5 minutes
3. **Slick Rick 1 (250 points):** Pick up a moisturizer power-up
4. **Slick Rick 2 (1250 points):** Pick up 5 moisturizer power-ups
5. **Slick Rick 3 (2500 points):** Pick up 10 moisturizer power-ups
6. **Slick Rick 4 (5000 points):** Pick up 10 moisturizer power-ups
7. **Powered Up 1 (250 points):** Pick up 1 power-up of any type
8. **Powered Up 2 (1250 points):** Pick up 5 power-ups of any type
9. **Powered Up 3 (2500 points):** Pick up 10 power-ups of any type
10. **Powered Up 4 (5000 points):** Pick up 20 power-ups of any type
11. **Roid Rage 1 (250 points):** Pick up 1 steroid power-up

12 **Roid Rage 2 (1250 points):** Pick up 5 steroid power-ups

13 **Roid Rage 3 (2500 points):** Pick up 10 steroid power-ups

14 **Roid Rage 4 (5000 points):** Pick up 20 steroid power-ups

15 **Speedy Streakzales 1 (250 points):** Pick up 1 energy drink power-up

16 **Speedy Streakzales 2 (1250 points):** Pick up 5 energy drink power-ups

17 **Speedy Streakzales 3 (2500 points):** Pick up 10 energy drink power-ups

18 **Speedy Streakzales 4 (5000 points):** Pick up 20 energy drink power-ups

19 **Turn on a Dime 1 (250 points):** Pick up 1 sneaker power-up

20 **Turn on a Dime 2 (1250 points):** Pick up 5 sneaker power-ups

21 **Turn on a Dime 3 (2500 points):** Pick up 10 sneaker power-ups

22 **Turn on a Dime 4 (5000 points):** Pick up 20 sneaker power-ups

23 **Press Space (1000 points):** Press the spacebar

24 **Enter the Principals Office (1000 points)**

25 **Enter the Cafeteria (1000 points)**

26 **Enter the Girls Locker Room (8008 points)**

27 **Enter the Basketball Court (1000 points)**

28 **Enter the Boy's Locker Room (1000 points)**

29 **Enter the Math Class (1000 points)**

30 **Enter the Biology Class (1000 points)**

31 **Enter the Lecture Hall (1000 points)**

32 **Traverse the hallway of DOOM (200 points):** Run all the way through the long narrow hallway on the top of the map

33 **Lost 1 Dumb Cop (1000 points)**

34 **Lost 5 Dumb Cops (1500 points)**

35 **Lost 10 Dumb Cops (2000 points)**

36 **Dastardly Escape x5 (1000 points):** Lose all cops after having 5 cops on your tail

37 **Dastardly Escape x10 (1500 points):** Lose all cops after having 10 cops on your tail

38 **Dead or alive, you're coming with me! (1000 points):** Activate the Robo-Cop

39 **Get Jiggy With it (2000 points):** Dance for 10 seconds total

40 **Put on a show (1000 points):** Dance for 5 seconds at once

41 **Superflash! x1 (1000 points):** Superflash 1 time

42 **Superflash x5 (2000 points):** Superflash 5 times

43 **Oh, the huge manatee! (2500 points):** Superflash 10 NPCs at once

44 **Exhibitionist x5 (1000):** Superflash 5 NPCs total

45 **Exhibitionist x15 (1500 points):** Superflash 15 NPC's total

46 **Exhibitionist x25 (2500 points):** Superflash 25 NPC's total

47 **Ramming Speed! x5 (1000 points):** Knock over 5 NPC's

48 **Ramming Speed! x15 (1500 points):** Knock over 15 NPC's

49 **Ramming Speed! x25 (2500 points):** Knock over 25 NPC's

50 **Ramming Speed! x50 (5000 points):** Knock over 50 NPC's

51 **Obstruction of Justice x5 (1000 points):** Knock over 5 Cops

52 **Obstruction of Justice x15 (1500 points):** Knock over 15 Cops

53 **Obstruction of Justice x25 (2500 points):** Knock over 25 Cops

54 **Obstruction of Justice x50 (5000 points):** Knock over 50 Cops

Artificial Intelligence

Finite State Machines & Decision Trees

Hierarchical finite state machines were used to implement NPC decision making. In the following images, blue ovals represent character states. White boxes represent what we call behaviors, which are essentially higher level states that include subsets of selected character states. When an NPC is in a particular behavior, he will always use a particular decision tree.

Behaviors

- Default - Characters starting behavior and go to state once he/she is unaware of the streaker.
- Aware - Characters are aware of the streaker.
- Knocked Up - Characters have been knocked over by being hit, run over or Superflashed.

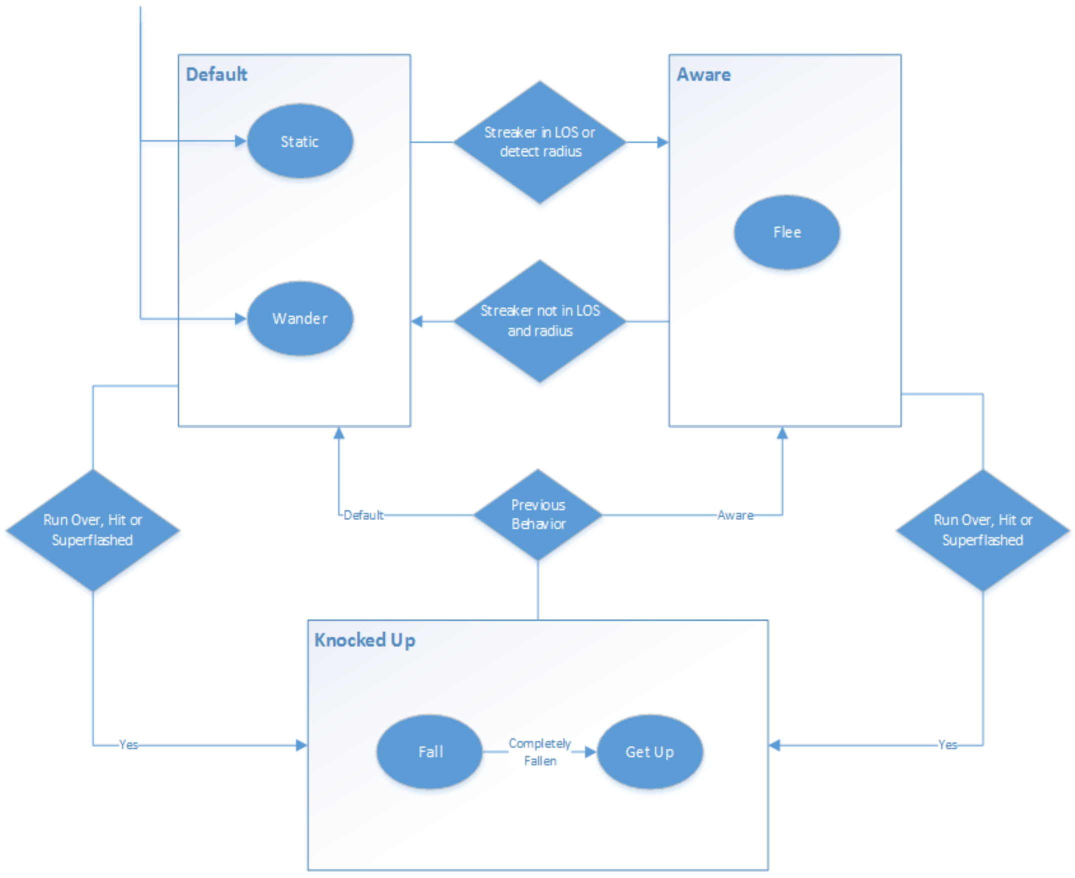
States

- Static - Character remains in place.
- Wander - Character wanders randomly.
- Seek - Character seeks streaker.
- Pursue - Character seeks streaker's future position.
- Path Find - Character follows optimized path to streaker. See path-finding section for more information.

- Fall - Character falls to ground and remains in its position.
- Get Up - Character gets up from ground and remains in its position.
- Flee - Character flees from the direction of the streaker.

Pedestrian

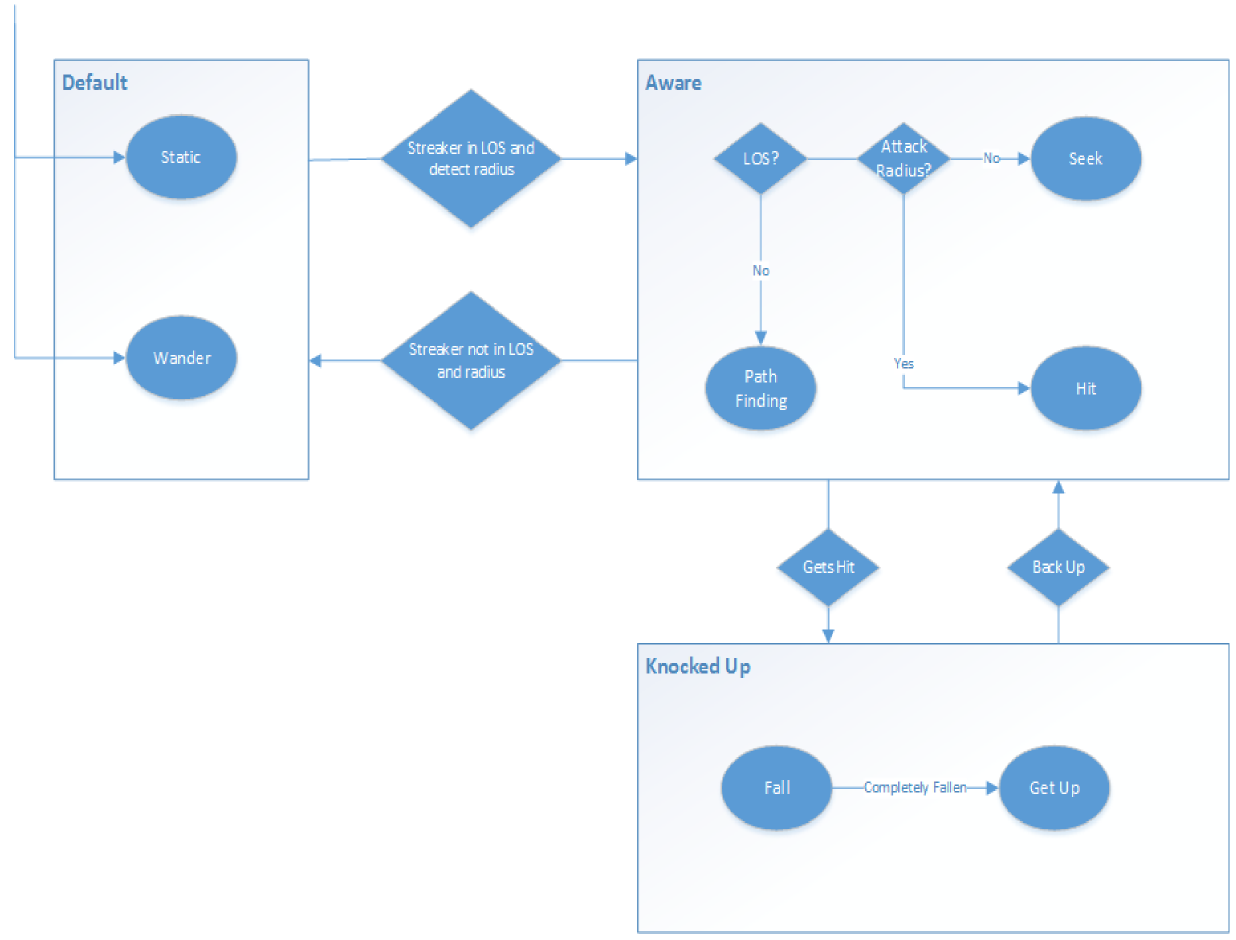
The pedestrian will either wander aimlessly or stand still until the Streaker is in sight and in range. At this point they will flee directly away from the Streaker until they can no longer see him or he is out of range and then return to its default behaviour. If the pedestrian is hit, it will fall, get up, and then return to its prior behaviour.



Default
Static
Wander
Streaker in LOS or detect radius
Aware
Flee
Streaker not in LOS and radius
Run Over, Hit or Superflashed
Default
Previous Behavior
Aware
Run Over, Hit or Superflashed
Knocked Up
Yes
Fall
Completely Fallen
Get Up
Yes

Dumb Cop

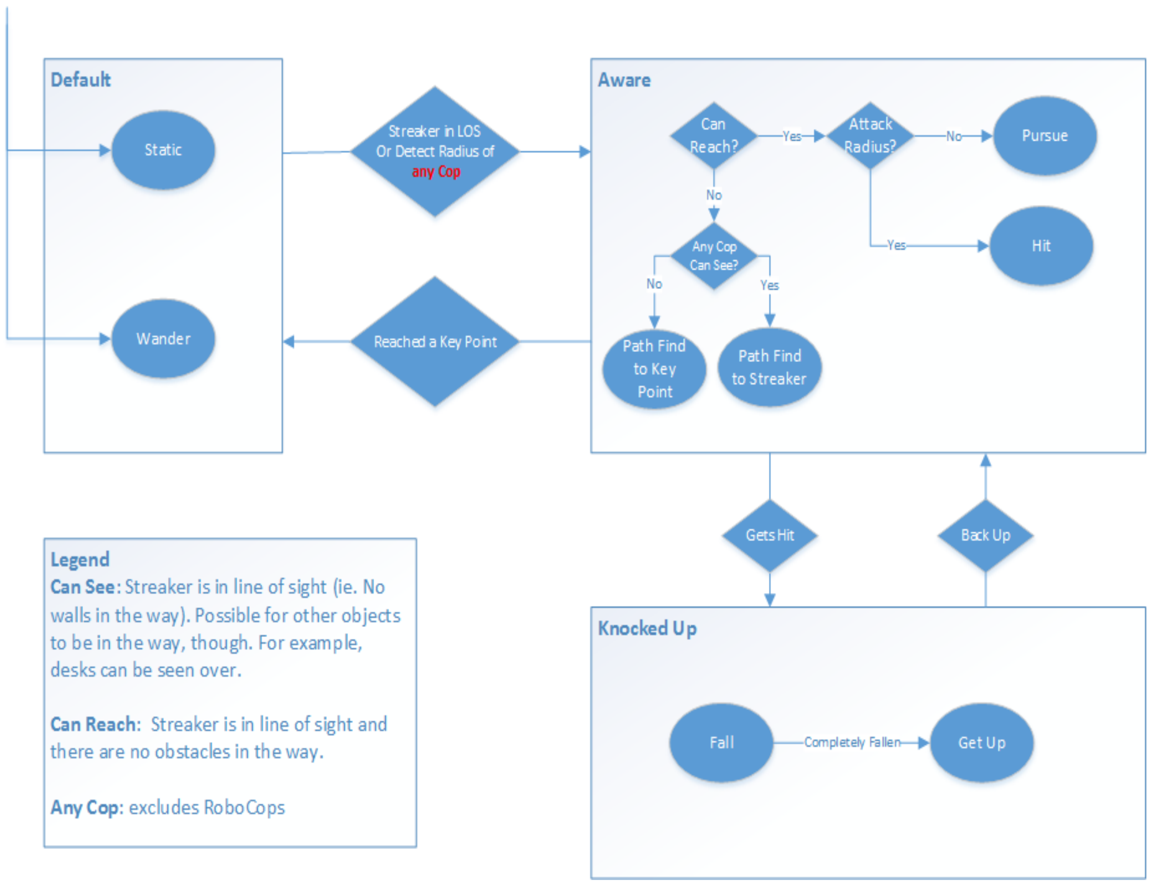
The dumb cop will only seek the Streaker when he is in sight and hit when he is in range. When the dumb cop cannot see the Streaker he path-finds to the Streaker's last known location and if he still cannot see the Streaker reverts to his default behaviour.



Smart Cop

Similar to the dumb cop, the smart cop will be aware of the Streaker if he is in sight and range. Additionally, he has the ability to gain awareness of the Streaker when any dumb or smart cop is aware of the Streaker. For certain obstacles, such as a desks, when the obstacle is in between a smart cop and a Streaker, then the cop can see the Streaker over the obstacle, but cannot "reach" the Streaker directly via his line of sight. In these scenarios, the smart cop either path-finds to the Streaker or to a key strategic position depending on if anyone knows his position.

On the other hand, if the smart cop can reach the Streaker directly, he will either pursue the Streaker or attack him, depending on if he is within an attack radius.



Legend

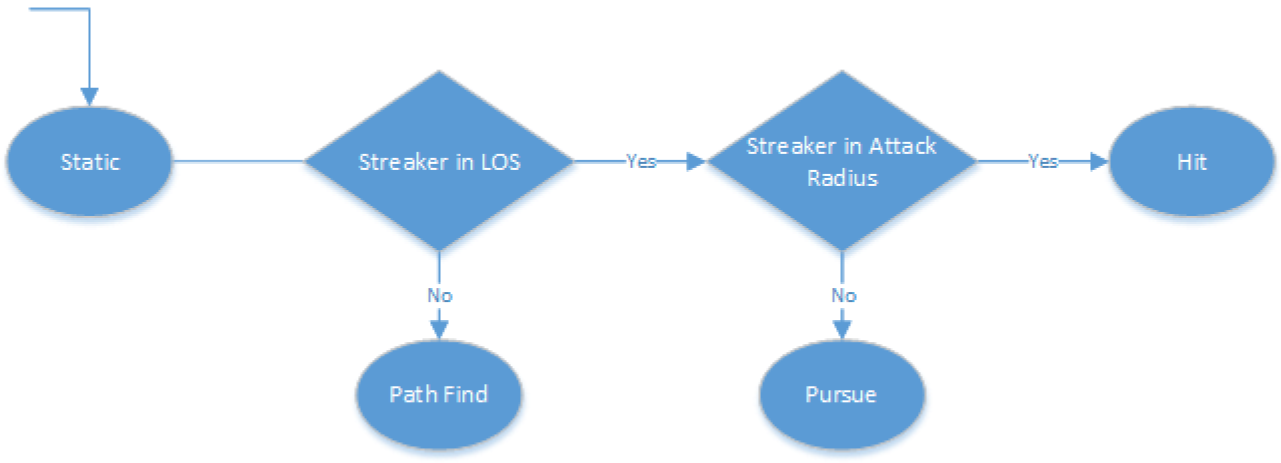
Can See: Streaker is in line of sight (ie. No walls in the way). Possible for other objects to be in the way, though. For example, desks can be seen over.

Can Reach: Streaker is in line of sight and there are no obstacles in the way.

Any Cop: excludes RoboCops

Robo-Cop

The Robo-Cop is static until he spots the Streaker. After this, he will pursue and path-find indefinitely. If the Streaker is in attack range, he will hit the Streaker.



Flocking

When a Smart or Dumb cop is spawned, it is assigned to a flock whose other members are the other cops of the same type. When the cop is calculating its velocity and position, it checks the flock as well. For every other cop within the flocking radius, it computes a cohesion vector toward the average position of the other members and a separation vector as the average direction away from the other members. These are then multiplied by the cohesion and separation weights respectively and added to the cop’s current velocity. As a result, nearby cops will flock together. This is very common when they are in pursuit of the Streaker and helps to keep them in a mob instead of lining up behind each other.

Path-finding

NPCs in the game path find using an implementation of the A* algorithm. The A* algorithm primarily runs in the standard way, using a List for the closed set and a Min-Heap data structure for the open set. Starting and goal nodes are chosen by finding the closest node to the desired positions, optionally checking that there are no walls in the way. The modified algorithm allows for weights to be optionally set on the path after it is computed. A maximum weight and a weight-per-node may be set. If this is done, the edge between the goal node and the penultimate node will have the maximum weight, the edge before that in the path will have the maximum weight minus the weight-per-node value, the edge before that will again have the weight-per-node subtracted, continuing until the start node. When the algorithm is next run, it adds the weights of the edges to the computed cost of the node. The effect of this is that if the algorithm is run repeatedly on similar start and end nodes, the best path may be seen as "occupied" if it has been calculated repeatedly and an alternate route will be chosen. Over time the weights of edges erode back to zero since after the path is traversed there is no longer a need for the weights.

After the A* algorithm is run and the path is returned, NPCs will check the position of the first node in the path and run to it. The NPC will also attempt to optimize the path by checking if the next node in the path is visible. If so, the first node in the path is removed and the next node, now first, is the NPC's new target. Nodes are continuously checked until the next node is no longer visible or there are only two nodes left in the path.

Because A* can be computationally expensive, some optimizations were made. If the algorithm is set to check for walls between the starting and ending positions and the starting and goal nodes, a QuadTree containing all of the walls is used to reduce the number of checks made. The heuristic used is also the square of the Euclidean distance instead of the actual distance since the square root operation can be slow. This way, the best path is not necessarily found, but almost always the path is “good enough”. Finally, the algorithm is only run when absolutely necessary. If an NPC can see reach its target, there is no need to path-find, so A* is not necessary. If the path must be continuously updated, as in the case of the Robo-Cop who continuously path-finds to the player when he is out of sight, A* is only run every five seconds instead of every frame. As a result the NPC may be path-finding to obsolete coordinates, but it is not long before it corrects itself.

Physics

Physics is a very important part of *Superflash*. Given the game's extremely stylized and arcade-like nature, the physics implemented did not have to provide a simulation level of realism. However, given the number of entities in the game and the collision-based nature of most of the mechanics, properly detecting and handling the collisions were very important.

The most basic and probably most important physical element in the game is player movement. Given that our game consists of running from and out-maneuvering running NPCs, it was necessary for character movement to look and feel good. For all characters in the game we employed steering behaviour using the standard three step Euler integration method as presented in the class slides. First, we set the acceleration of the player to be in the player's determined direction at a magnitude equal to the character's maximum acceleration. Using this acceleration and the time gone by since the last update, we calculate the character's new velocity and limit it based on the character's maximum velocity. Finally, we update the character's position using the same time value and the new velocity value. By modifying the maximum acceleration and velocity values, we were able to obtain realistic looking character movement which also felt natural when controlling the streaker.

A second very important physics-based component of our game is collision detection. From the very beginning of the project, we wanted to create a frenzied atmosphere by including as many NPCs as possible in our game. Given the collision-based nature of the game, ignoring collisions between different entities was not an option. However, we knew that checking for collisions between all pairs of entities in the game would be much too slow. To solve these problems, we used a few optimized collision detection techniques.

First, we used a modified sweep and prune method for collisions between moveable entities. At the start of the game, all moveable entities are placed in two lists, one which sorts entities based on their X position and one which sorts entities based on their Y position. At every update, these lists are sorted, which is can be done very quickly given that most entities will not move enough in one update to swap positions with any other entities. Once these lists are sorted, the lists are scanned and collision tests are only performed between elements whose differences in position are less than the summed radiuses of their bounding shapes. The result of this technique is that collision checking between moving entities is very quick, so that even when the map is full of NPCs there is no noticeable slowdown.

The second optimization to collision detection is done using a grid to store walls in the map. The grid is a two dimensional array of lists, each of which contains a copy of any obstacle which intersects with the corresponding of 200 x 200 pixel tile. At the beginning of the game, each obstacle in the level is placed in the lists of all corresponding grid positions. In the game's update, collisions are then tested for between each moveable object and the obstacles which map to the same grid positions that they do. This results in very quick collisions checks between moveable objects and all static obstacles.

For simple collision detection, bounding rectangles were the only shapes used in the game. All objects were tightly fitted with bounding rectangles, and characters had bounding rectangles placed at their feet (to properly simulate collision given the game's perspective). While we considered using bounding circles, the shapes in our game worked better with rectangular collision detection. Bounding rectangles allowed for simple collision checks with better fitting than bounding spheres. Given the rectangular shapes of the objects in our game, the bounding boxes are axis aligned and move with the character rather than being recreated at every update.

For collision resolution, the steps we followed were those presented in the class with one small modification. All objects in our game could easily be surrounded with bounding rectangles except for NPCs. Although the game's perspective makes the characters look flat, collision would seem very unnatural if the entities were treated as rectangles, but putting a sphere or ellipsoid shape around their feet would complicate collision detection besides making it more difficult to properly surround the player's feet. To fix this, we used bounding boxes for all collision detection, but handled resolutions with NPCs by treating the bounding rectangles like circles. As such, given a rectangular intersection, we would find the distance from the intersection point to the center of the rectangle and treat that value as the inter penetration distance subtracted from the circle's radius. Collisions and inter-penetration were then resolved using the circle steps outlined in the slides. The result of this hybrid method was that in addition to collisions being easy to detect and resolve, collisions between characters were more realistic and conveyed a sense of the characters' roundness to the player. In order to ensure that collision resolutions looked even more realistic, low coefficients of restitution were used for all moveable entities.

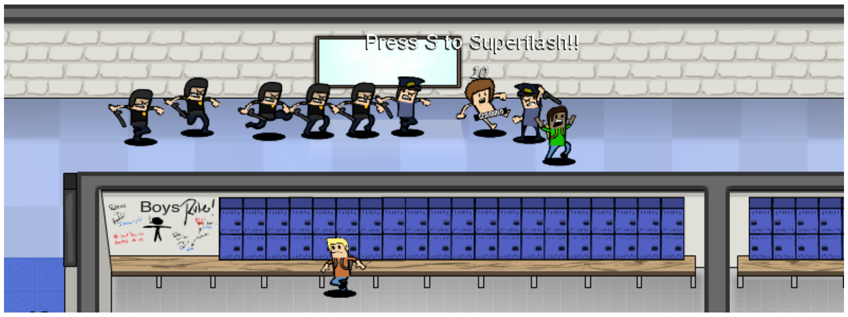
Results

In order to ensure that the game worked properly we primarily focused on play-testing. By repeatedly playing the game we became very familiar with how the game ran and if any bugs appeared they were immediately obvious. Play-testing was also the means by which we balanced the game. Continuous tests allowed us to determine good values for subjective criteria, such as the running speed of the player and NPCs and the points given for actions.

Certain bugs that were very difficult to remove were instead checked for and handled. For example in rare cases it is possible for an NPC to be pushed out of the level, causing an exception to trigger when invalid grid coordinates are used. As this deals with a very complex part of the game, it was decided that the exception should be checked for and handled by either ignoring it or, if the exception was occurring in the world's update, to remove the offending NPC altogether. The NPCs are numerous enough that losing one is not noticed by the player and a lot of time is saved by not having to try to fix was is a common and complex issue, even with big games.

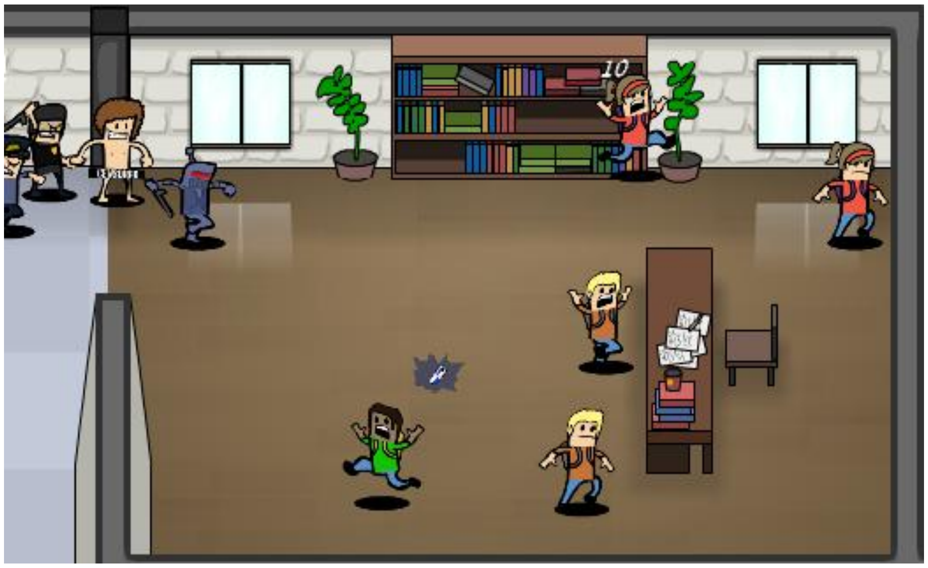
The resulting game, though pared down from our team's initial ambitions is still extremely satisfactory. Especially pleasant were the unexpected behaviors caused by many interacting pieces of the game. For example, the narrowness of the top hallway combined with the sheer number of enemy NPCs chasing the player meant that it is a very difficult hallway to traverse, and so we added an achievement acknowledging this.

An ambush in the Hallway of DOOM



Also, we decided purely as a concept that the energy drink should be placed in the principal's office. Through play-testing we discovered that the small size of the room and the single exit made it a very dangerous room to enter. This ended up working out very well, as we also determined through play-testing that the speed boost provided by the energy drink was the most useful ability boost, making the energy drink the most valuable power-up. Accidentally pairing the two created a very satisfying risk/reward scenario for the player. We further emphasized the risk by having the first Robo-Cop spawn in this room.

The office with its risk and reward



Overall, we are satisfied with the way the game plays and genuinely enjoy playing it. The game is fast, hectic and the score system makes it very competitive. The best representation of the game play is given by the available YouTube video[4].

[4] http://www.youtube.com/watch?v=BYDV6YHAt2U

User Manual

Compilation

The project is compiled by obtaining the latest version of the code from the repository[5] and running the solution file in Microsoft Visual Studio 2010 (C#). The user must also first download the Microsoft XNA Game Studio 4.0 in order to compile the game. It is available from the Microsoft website[6].

The game is intended to be built in Release mode. Building in Debug mode will cause the game to run windowed instead of in full-screen and all bounding boxes at path nodes will be visible.

Playing the Game

When the game start the player is presented with a simple menu. The game is started by clicking the "PLAY" button. Once the game is started, the Streaker is controlled using the keys and buttons described in the Controls section of this document. By default the keyboard is used. If an Xbox 360 controller is detected, the keyboard is disabled and the controller is used instead. The player must run around the level attempting to gain as many points as possible while avoiding enemy police officers. Points are gained by chasing and knocking down pedestrians, dancing, using the Superflash and by earning achievements. When an enemy reaches the player, it will attempt to hit the player. When the player is hit, health is lost. When the player runs out of health, the game is over. From the Game Over screen, the player can

[5] http://code.google.com/p/comp-476-project/source/checkout
[6] http://www.microsoft.com/en-ca/download/details.aspx?id=23714

then click either the “Replay” button to restart the game or the “Main Menu” button to return to the main menu.

Level Editor

The user is initially presented with an empty level. The user may load the contents of “level.txt” from the StreakerLibrary folder before the project is built by pressing the L key. The level’s graphic serves as a background. The user may place objects by using the keys outlined in the Controls section of this document. Every object in the level is represented graphically. Boxes are either red if they are walls or orange if they are obstacles that can be seen through (e.g. desks, tables). Path-finding nodes are blue circles with lines between them as edges. NPCs are depicted as they are in the game, with color-coded circles marking the hit area for deletion and a letter indicating their initial behavior. Triggers are yellow boxes with a number denoting their ID. Consumables are depicted with the same icons as in the game. The player is represented as it is in the game with an orange circle for the hit area for deletion.

Boxes and triggers are placed by entering Box or Trigger mode then clicking and dragging with the mouse. The endpoints of this drag will be opposite corners of the box or trigger to be placed. Right-clicking a box or trigger will delete it from the level. Nodes are placed by entering Node mode and clicking in the map. After two or more nodes are placed, the user can create edges between nodes by clicking one node and dragging the mouse to another node. Nodes can be deleted by right-clicking them, which will also delete associated edges. NPCs are placed by entering NPC mode, selecting the type of NPC to place and the default behaviour, then clicking on the map. NPCs can be deleted by right-clicking in their circle. Consumables are placed by entering Consumable mode, selecting the type of consumable and clicking on the map. Consumables are deleted by right-clicking them. The player is placed by simply clicking on the map. Only one player may be placed in the level.

When the map is ready, it can be saved by pressing the S key. This will save all of the information to “out.txt” in the level editor’s working directory. To load the new level in the game, the user must copy the contents of “out.txt” and replace the contents of “level.txt” either in the StreakerLibrary folder before building the project or in the working directory of the game after building the project.