Soft/Rigid Bodies in Pacman

Team 1
Ayush Kharade 40042388
Christian Plourde 26572499
Daniel Vellucci 27416288
Lyonel Zamora 27385986
Samer Yazbeck 40049573

December 2019

Contents

1 Introduction **3**
1.1 Overview . . . 3
1.2 Team Contributions . . . 3
1.3 Animation Techniques . . . 3
1.3.1 Soft Body Cloth Simulation . . . 3
1.3.2 Collision Detection . . . 4
1.3.3 Particle Systems . . . 4
1.3.4 Motion Capture from Kinect . . . 4
1.3.5 Ghost AI . . . 5
1.4 Deployment . . . 5

2 Methodology **6**
2.1 File Hierarchy . . . 6
2.2 Class Diagram . . . 7
2.3 Source Files . . . 8
2.3.1 Mocap . . . 8
2.3.2 Vectors . . . 9
2.3.3 Cloth Particles . . . 10
2.3.4 Cloth Constraints . . . 13
2.3.5 Cloth . . . 14
2.3.6 Game Board . . . 19
2.3.7 Ghosts . . . 21
2.3.8 Pacman . . . 33
2.3.9 Main.cpp . . . 33
2.3.10 Audio . . . 34
2.3.11 Delta Time . . . 35

2.3.12 Pacman Collision 35
2.3.13 Trail Particles 35
2.3.14 Particle System 37
2.3.15 Breakable Walls 37
2.4 Use Cases 39
2.4.1 Scenario 1 39
2.4.2 Scenario 2 39
2.4.3 Scenario 3 39
2.4.4 Scenario 4 39
2.4.5 Scenario 5 39
2.4.6 Scenario 6 39

3 Conclusion **40**
3.1 Limitations and Challenges 40
3.1.1 Performance 40
3.1.2 Camera Controls 40
3.1.3 End game 41
3.1.4 Overlap grid collision 41
3.1.5 PowerUp 41
3.2 Scalability 42
3.2.1 More maps 42
3.2.2 More ghosts 42
3.2.3 Additional Controls 42

4 References **43**

1 Introduction

1.1 Overview

The goal of this project was to recreate the arcade game Pacman with a more modern twist using advanced animation techniques to give it a unique feel. We started working on this project by using a base Pacman OpenGL code that we found that was unfinished [see ref 1] (didn't have PowerUps or Ghost Collision) and added to it many things such as cloth simulation, particle systems and motion capture based controls to give it a unique feel. The game plays the same with the goal being to collect all coins without dying at the hands of the ghosts.

1.2 Team Contributions

The team's major additions from the base Pacman [see ref 1] we started with are:

- Kinect controls for Pacman
- Cloth simulation for the ghosts with collision detection between cloth and ball
- Particle trail following both Pacman and the ghosts
- Particle explosion on PowerUp pickup with ghost color change and implementation of PowerUp mechanics
- Particle on wall break and wall break mechanics
- Collision detection between ghost and Pacman
- In game sounds

1.3 Animation Techniques

1.3.1 Soft Body Cloth Simulation

The traditional ghosts in the Pacman game are created in this project using cloth simulation inspired from the tutorial from [see ref 2].

- A grid of particles which are linked to neighboring particles to simulate cloth using a spring mass system.
- This cloth is instantiated as a plane, on top of a sphere.
- Using gravity and collisions on all particles in the cloth, the cloth covers the sphere, which is used to depict a ghost for our Pacman project.
- The cloth is a child of the sphere, so it follows the sphere around the map.

1.3.2 Collision Detection

For this project we implemented 2 types of collision detection.

The main collision detection is a grid-based collision detection that is used to detect collision between PacMan and walls, ghosts, coins and cherries. This is also used to detect collisions between ghosts and walls. This works since our map is a grid based map with an X,Y coordinate representing each tile on the board. This allow us to check if two objects exist on the same tile. If this is the case, then they collide, which causes one of the following scenarios:

- If player hits Ghost - Player loses a life
- If player hits Coin - Total coin count decreases
- If player hits Cherry - PowerUp activated and particle explosion animation plays
- If player hits Breakable Wall - Wall disappears and a particle effect appears

The second collision detection is used between the cloth and the sphere it sits on, using a bounding sphere between the sphere itself and each particle in the cloth. When a collision occurs the particle gets pushed back to the outside of the sphere.

1.3.3 Particle Systems

We used a multitude of particles in our game which we made by taking inspiration from both [see ref 3] and [see ref 4].

- Breakable walls separate into smaller bricks when collided with. Upon collision, each brick goes in a separate direction
- Trails are particles that spawn at player/ghost position as they move and dissipate with time giving the illusion of a trail
- PowerUps create an explosion of particles that flow outwards towards the screen when picked up

1.3.4 Motion Capture from Kinect

Motion capture can be used as an alternative control model to control the player and play the game. This is implemented using the Windows Kinect SDK V2.0 [see ref 5] and using the tutorial from [see ref 6].

- Motion capture allows the player to control Pacman with his/her right fist
- Mimic up, down, left and right motions with your fist while in front of the Kinect to control the player movement
- Player will only change direction if there is a possible route to go to

- Player can still be controlled with the keyboard controls even if the Kinect is on

More details on how to play with the Kinect can be found in the User Manual.

1.3.5 Ghost AI

The ghosts can have two different types of AI to guide their movement.

First, ghosts can be of type “Tracker” which will pursue the player if Pacman is in a range. This is done by comparing grid positions of the ghost and Pacman. This is used to find which direction the ghost should travel in order to reach Pacman. The ghost then moves in that direction if possible.

Second, ghosts can be of type ”None” which will make him move in random directions when passing intersections.

1.4 Deployment

For this project to work you will need to start by downloading Windows Kinect V2.0 from the windows website [see ref 5]. This only works on a Windows Operating System

When that is downloaded you can clone the following repo:
https://bitbucket.org/comp477f19team01/comp477-f19-01/src/master/

Run this code in Visual Studio (preferably 2019). If another version of Visual Studio is used a re-targeting of the solution may be required.

IMPORTANT NOTE: It is possible that when building the code (even in Debug mode) that the SFML dll cannot be found and thus compilation fails. To rectify this, within the project folder, copy all the files ending in .dll from the SFML folder into the Release folder.

2 Methodology

2.1 File Hierarchy

The project contains a Visual Studio Solution that separates the code into two important folders called the header and source files folders. An example is shown in Figure 1 below, which illustrates the separation between the Header Files and Source Files into their respective folders. While some of the files, namely the main.cpp file which contains the main function, do not have a header file associated to them and thus don't have anything in the Header Files folder. The relation between header files and source files is shown in Figure 2, to give a better understanding of what files are contained in the source code and how they work together.

Figure 1: File Hierarchy

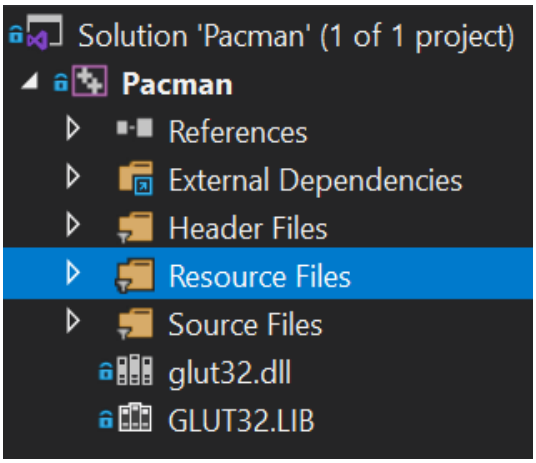
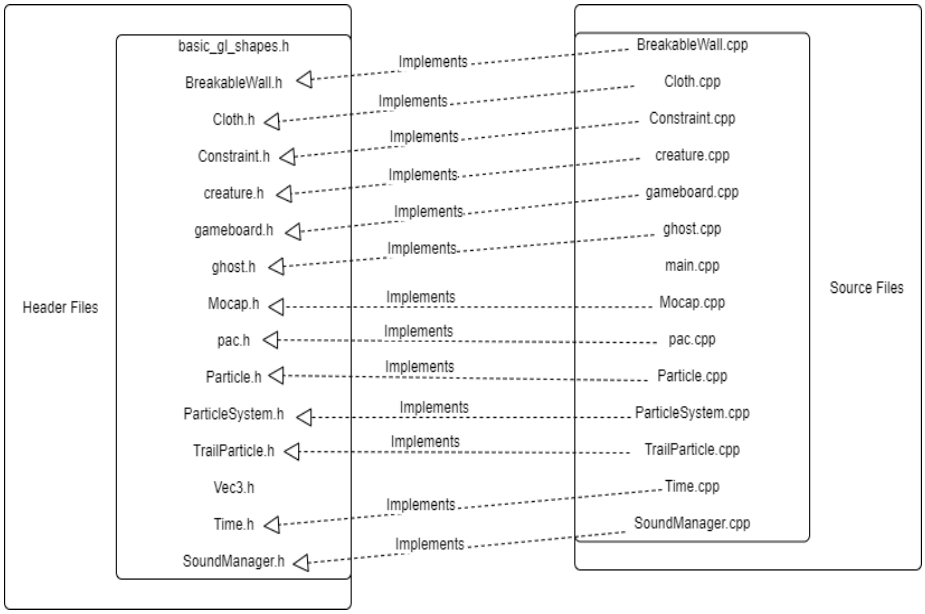


Figure 2: Header to Source File Mapping



2.2 Class Diagram

A diagram showing the classes that make up the architecture of the system is shown in Figure 3.

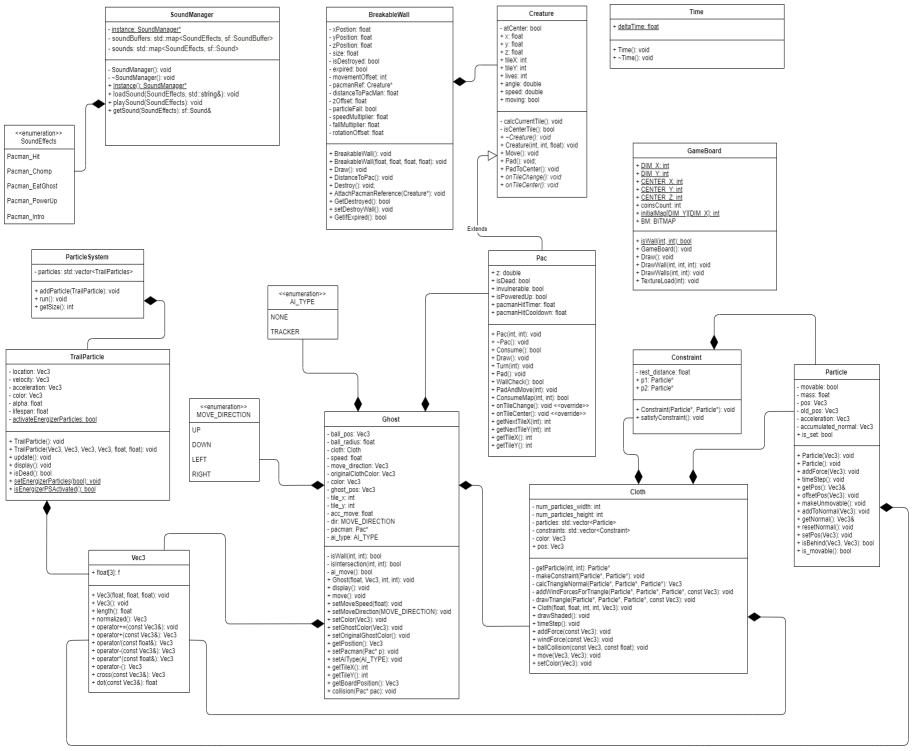
As can be seen from the diagram, a very important class is the Vec3 class, which is a data structure used to store a vector with three elements and perform vector algebra. This class is used by the particle system used to make the trails (ParticleSystem class) as well as the Ghost class and the Cloth class to name a few.

The pacman class inherits from the creature class which contains basic information about creatures such as their position on the tile grid indicated by the tileX and tileY fields as well as their movement speed and a boolean to indicate if the creature is moving or not. In the original version of Pacman that we used as a starting point [1, see ref], the ghosts also extended from this class.

The Ghost class is also composed of two enumerator types. The first, AI_TYPE, describes the type of AI used for determining the ghosts movement as explained in Section 1.3.5. The second, MOVE_DIRECTION, indicated the direction of motion of the ghost (UP, DOWN, LEFT, RIGHT). This makes

the code easier to read and makes switching between cases for movement easier than using a vector to describe the direction of motion, even though the ghost class does have a vector describing its direction of motion (move_direction) to describe its motion for use in calculating its new position from frame to frame.

Figure 3: Class Diagram



2.3 Source Files

2.3.1 Mocap

The mocap.cpp, whose location in the file hierarchy can be seen in Figure 2, is where all Kinect control happens implemented using the following tutorial [see ref 6]. It gets called by the main function using the launchKinect function which takes argc and argv but also the pacman object.

This function (launchKinect) creates a new window and initializes the Kinect for that purpose. The new window will display a point cloud map with depth and RGB for each point as well as a red dot when it detects your right hand

and a green squared centered in the window.

The Output function is the function that checks if the hand (red dot) position left the square and if so what direction and acts accordingly (move pacman in the right direction).

The draw function is the display and idle function for the glut main loop of this window.
-It draws the point cloud using the getKinectData function which also gets the body data (the getkinectdate function calls the following function each getting its respective data: getBodyData, getRgbData and getDepthData).
-It also draws the square and the point representing the hand once its able to track a human in the feed.

The initKinect, initializes the Kinect if there is one connected.

This file mainly uses Glut and Kinect.h as dependencies. The latter being brought from the Kinect Windows SDK v2.0. This file has one main data structure that is the joints vector that holds all the joints detected by the Kinect in the getBodyData function.

2.3.2 Vectors

The project uses only one type of vector, a vector with three elements, and this data structure is defined in Vec3.h, whose location in the file structure can be seen in Figure 2. The class has one data member, which is an array of three floats containing the x, y, and z components of the vector. This array is public and so its components can be modified freely.

The Vec3 class allows many operations often used in vector algebra such as cross product which is defined by the formula shown in Equation 1.

$$\overrightarrow{A} \times \overrightarrow{B} = \begin{bmatrix} A_1 \times B_2 - A_2 \times B_1 \\ A_2 \times B_0 - A_0 \times B_2 \\ A_0 \times B_1 - A_1 \times B_1 \end{bmatrix} \tag{1}$$

The class also allows for dot product of two vectors which is defined in Equation 2.

$$\overrightarrow{A} \cdot \overrightarrow{B} = A_0 \times B_0 + A_1 \times B_1 + A_2 \times B_2 \tag{2}$$

The other operations defined are simple vector operations such as subtraction, addition, multiplication and division by scalars, etc.

The class also has a length function which returns the magnitude of the vector. This is calculated using Equation 3.

$$|\overrightarrow{A}| = \sqrt{A_0^2 + A_1^2 + A_2^2} \tag{3}$$

Finally the class allows for normalization of the vector, which is useful in determining motion directions since this vector will have length one and can be scaled more appropriately, as well as in the determination of forces in the cloth simulation portion of the code described in Section 2.3.5. The equation is shown in Equation 4.

$$\overrightarrow{A_{norm}} = \frac{\overrightarrow{A}}{|\overrightarrow{A}|} \tag{4}$$

2.3.3 Cloth Particles

A cloth consists of a network of particles, which are defined in the Particle.h file, whose location in the file structure can be seen in Figure 2.

The particle class has many data members, whose functions are enumerated as follows:

movable A boolean indicating if the particle should be movable, i.e. it can be displaced when the cloth moves or forces are applied to the cloth. This can be used to anchor certain particles in the cloth. In the one used for the ghost, the middle particle in the cloth is unmovable and place at the center and top of the sphere, allowing the cloth to drape around the sphere. See Section 2.3.5 for more information.

mass A float indicating the mass of the particle. This is required to compute and add forces to the particle using Newton's Second Law shown in Equation 5, where m is the mass of the particle, $\overrightarrow{a}$ is the acceleration vector, and $\overrightarrow{F}$ is the resulting force vector.

pos A Vec3 as defined in Section 2.3.2 indicating the position of the particle in world space. This is required to actually render the particle as well as compute its movement and compute forces.

old_pos A Vec3 as defined in Section 2.3.2 indicating the position of the particle in world space in the previous frame. This is required to compute the new position in the timestep function by doing Verlet Integration. This is done with Equation 6, where $\overrightarrow{pos}$ is recomputed by taking the difference between it and the $\overrightarrow{old_pos}$ and multiplying it by a damping factor D so that the force applied to it decreases in effect over time and then added to the acceleration vector $\overrightarrow{a}$ multipied by a timestep factor TS to scale it with the frame rendering rate.

acceleration A Vec3 as defined in Section 2.3.2 indicating the acceleration of the particle in world. This is set when a force is added to the particle and

is used in the Verlet Integration to calculate the new position based on the force applied.

accumulated_normal A Vec3 as defined in Section 2.3.2 indicating the normal of the triangle that this particle is contained in in the cloth. This is set when constructing the cloth as triangles are created with triplets of particles. The normal of each particle in the triangle is added to the particles normal to get a global normal for that triangle that can be used in calculating the wind force on a triangle in the cloth.

is_set A boolean indicating if the particles position is set or not. This is used in the collision detection of the cloth and the sphere in the cloth class defined in Section 2.3.5. If the position has been reset by the collision, i.e. the particle has been moved back to the surface of the sphere so it does not clip through it, we set this to true so that the position is not reset in the frame.

$$\overrightarrow{F} = m \times \overrightarrow{a} \tag{5}$$

$$\overrightarrow{pos} = \overrightarrow{pos} + (\overrightarrow{pos} - \overrightarrow{old\_pos}) \times (1 - D) + \overrightarrow{a} \times TS \tag{6}$$

The particle class has functions that allow for addition of forces as well as computation of position of the particle and some utility functions. These are described below.

Particle() A default constructor that simply sets the is_set data member to false. This is a utility constructor and is not used in the cloth simulation.

Particle(Vec3 pos) A constructor that takes in a position vector that indicates where the particle should be created in world space. This will initialize the pos data member to this position, as well as old_pos since they are initially equal as no force has yet been applied. The mass of the particle is set to 1 (in this code each particle has equal mass). Each particle is also set by default to moveable since this will be true for each particle except one, which is made unmovable when creating the cloth.

addForce(Vec3 f) A function that adds a force to the particle by passing in a Vec3 that contains a force vector. The acceleration data member of the class is set by using Newton's Second Law as described in Equation 5 in reverse to get the acceleration, which is then set and can be used in the Verlet Integration step to compute the new position after the force is applied.

timeStep() A function that is called once per frame and recomputes the position of the particle based on its acceleration. It does a step of the Verlet Integration if the particle is movable using Equation 6. Once this is done the old_pos is reset to the current pos so it can be used in the next integration step and the acceleration vector is set to 0 since it has been applied to the particle.

getPos() A function that returns the position of the particle.

offsetPos(Vec3 v) A function that will move the position of the particle by a step of v. For example if the position of the vector before calling the function was (1, 0, 1) and v is (2, 0, 3), the new position after calling the function will be (3, 0, 4).

makeUnmovable() A function that makes the particle unmovable. This means that the particle is no longer affected by forces.

addToNormal(Vec3 normal) A function that adds the passed in vector to the particles accumulated_normal and places the result in the accumulated_normal data member. This is used to increment the accumulated normal for use in calculating wind forces on the cloth as explained previously in the description of the relevant data member. It is important to note that whatever is passed in to the function is normalized before aggregating it with the accumulated_normal.

getNormal() A function that returns the accumulated_normal.

resetNormal() A function that sets the accumulated_normal to the zero vector.

setPos(Vec3 pos) A function that sets the position of the particle to a position in world space.

isBehind(Vec3 move_speed, Vec3 check_pos) A function that checks if a particle is behind a specific position. This is done by taking the vector difference between the particles current position and the "next position", which is the particles current position vector summed with the move_speed. This will give the direction of motion of the particle. Next we compute the vector difference of the particles position and check_pos, which gives the direction to the position we want to verify. By taking the dot product of these two resulting vectors, the sign will tell us if the particle is behind (a negative dot product indicates that the particle is behind the check_pos). This is useful when computing forces. If we wish to only apply forces to particles behind a certain position, say the center of the sphere, we can make the cloth look like it is trailing behind the ghost while the

particles in front of the sphere are unaffected (or to a much lesser degree).

is_movable() A function that returns a boolean indicating whether or not the particle is movable.

2.3.4 Cloth Constraints

Constraints on the cloth are required in order to properly model a spring mass system. For example, a particle has a limited range to which it can move from its neighbors before springing back towards them. This simulates the stretching of threads in a real piece of cloth. The class is defined in Constraint.h, whose location in the file structure can be seen in Figure 2.

A constraint is defined for a particle pair and is based on a "rest distance". This is the original distance between the pair of particles. Any deviation from this rest distance will be corrected in this class iteratively so as to simulate the motion of a spring.

The data members of the class are explained in detail below.

rest_distance A float indicating the original distance between the particle pair that the constraint applies to.

p1 A pointer to the first particle in the pair that the constraint applies to.

p2 A pointer to the second particle in the pair that the constraint applies to.

The functions of the class are explained in detail below.

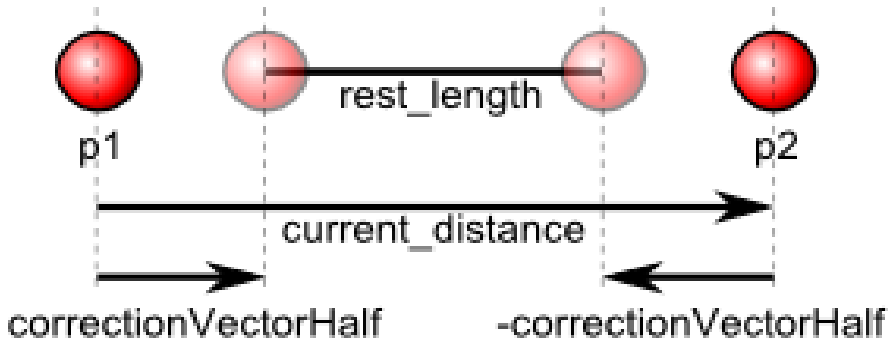
Constraint(Particle* p1, Particle* p2) A constructor that takes in two pointers to particle objects. It sets the pointer data members of the class appropriately and then computes the rest_distance by taking the length of the vector resulting from the vector difference of the positions of the two particles.

satisfyConstraint() This functions attempts to rectify the position of particles that have stretched beyond or under their rest length. This situation is illustrated in Figure 4 taken from [see ref 2]. In order to rectify the situation, we compute the correction Vector as illustrated in the figure. To do this, we take the ratio of deformation from the rest_length using the formula shown in Equation 7. This yields a vector that will correct the position of the particles p_1 and p_2 in a way that is proportional to how much they have been stretched from their resting positions. The division by 2 occurs due to the fact that each particle takes on half of

the total correction. Once the $correctionVectorHalf$ has been computed, we offset the position of p_1 by adding that vector to it to get its new position and we subtract the same vector from p_2 since it should move in the direction opposite to that of p_1 to return to its resting position. This function is called iteratively since the constraints must be satisfied many times to create a smooth looking motion. If it is called once, the particle will return to the position outside its rest_length once and then not move which does not produce the desired effect. More details on this can be found in Section 2.3.5.

$$correctionVectorHalf = \frac{(p_2 - p_1) * (1 - \frac{rest\_length}{|p_2 - p_1|})}{2} \quad (7)$$

Figure 4: Cloth Constraint Satisfaction



2.3.5 Cloth

This class is the class that represents the complete cloth model by combining a network of particles and constraints. This class contains the rendering methods, as well as the methods that actually use all of the tools created in the Constraint and Particle classes defined in Section 2.3.4 and Section 2.3.3 respectively.

The data members of the class are explained in detail below.

num_particles_width An integer that dictates how many particles wide the cloth should be.

num_particles_height An integer that dictates how many particles high the cloth should be.

particles A vector container (from the c++ standard library) that contains all of the particles that are part of the cloth. Allows for easy iteration through the particles for satisfying constraints.

constraints A vector container (from the c++ standard library) that contains all of the constraints that are part of the cloth. Allows for easy iteration through the constraints for satisfying them.

color A Vec3 that dictates the color of the cloth.

pos A Vec3 that dictates the position of the cloth. The position of the cloth is determined by the position of the middle particle (half width and half height). The other particles are placed with respect to this position.

The functions of the class are explained in detail below.

getParticle(int x, int y)

This function returns a Particle pointer that corresponds to the particle at position (x, y) on the grid that the cloth consists of. For example getParticle(3, 4) would yield the particle in column 3 and row 4. Since the particles are stored in a single dimensional vector, this is done by taking the y value and multiplying it by the width of the cloth (num_particles_width) and then adding x as the offset. This makes sure we are in the right "row" and then proceeds to move to the correct "column".

makeConstraint(Particle* p1, Particle* p2)

This function creates a new constraint based on the two Particle pointers passed in and pushes it into the constraints vector (the data field). This function is called in the constructor of the cloth many times to make sure that all the constraints, i.e. relations between neighboring particles have been created properly.

calcTriangleNormal(Particle* p1, Particle* p2, Particle* p3)

This function computes the normal of the triangle made from the three passed in particles' positions. This is done by computing the vector from p1 to p2 and p1 to p3. Taking the cross product of these two vectors will yield the normal of the triangle contains these three particles.

addWindForcesForTriangle(Particle* p1, Particle* p2, Particle* p3, Vec3 direction)

This function is used to simulate a wind force acting upon the cloth. This function should be called for each triangle in the cloth seeing as it acts on triangles and not the entire cloth. It works by first getting the normal of the triangle

that contains p1, p2, and p3 by using the previously defined function. Once this is known, the force that should be exerted on each particle is determined by taking the dot product of the normalized normal of the triangle the direction of the wind force. This is then multiplied by the actual normal to rescale it (resulting in a force vector) that is then applied to each of the three particles in the triangle using the addForce method of the Particle class described in Section 2.3.3. This ensures that the maximum force will be applied when the wind is blowing perpendicular to the triangle created by the three particles, and 0 when the wind is blowing parallel to the triangle.

drawTriangle(Particle* p1, Particle* p2, Particle* p3, Vec3 color)

This function renders the triangle described by particles p1, p2, and p3 in the appropriate color (defined by a Vec3 in RGB format). To do this, the function glTranslatef is used to position ourselves at the position of the cloth (its center in this case). We then switch the color to the appropriate, and normalize the normals of each Particle. We then send this normal data to OpenGL to draw as well as vertex data for each Particle, which is essentially the particle's position.

Cloth(float width, float height, int num_particles_width, int num_particles_height, Vec3 pos)

This constructor creates the entire structure for the cloth and sets the constraints, as well as the positions of all the particles. It is the most important function in the class. The width and height here are the width of and height of the cloth. This is different from num_particles_width and num_particles_height as these dictate how many particles are in the cloth, and not its actual size. More particles but constant width will result in a more realistic cloth at the cost of more computation.

The first thing we do is to resize our vector container of particles to have enough space to hold all of the particles. This means the vector now has a size of $num\_particles\_width \times num\_particles\_height$, these being set to the values of num_particles_width and num_particles_height passed in to the function.

Next, using a double for loop over the particle grid, we instantiate the particles at the correct positions. To compute the position of the particles, we create a Vec3 that has a position defined by the position of the cloth (the middle of it) and an offset based on the x and y position of the particle in the grid of the cloth. The formula is shown in Equation 8. Here $\overrightarrow{Pos}$ is the position of the particle for the current iteration in the double for loop, in other words, the position of the particle at position (x, y) in the grid of the cloth. In these equations, the pos variables represent the position of the cloth, i.e. its center.

Essentially we determine where the particle should be with respect to the middle of the cloth by determining how far along the width of the cloth we are and the offsetting that by the position of the cloth itself. The reason for the negative sign when computing the y coordinate of $\overrightarrow{Pos}$ is due to the fact that as we go down y actually increases (a simple convention for the coordinate system of the grid). The z value of the position remains unchanged since the cloth lies in a since plane.

$$\overrightarrow{P} = \begin{bmatrix} width \times \dfrac{x}{num\_particles\_width} + pos_x \\ -height \times \dfrac{y}{num\_particles\_height} + pos_y \\ pos_z \end{bmatrix} \tag{8}$$

The next step is to create the constraints between the particles using the makeConstraint method of the class. Essentially, each particle will have at most 8 constraints, in the situation where the particle is within the middle of the cloth, by creating a constraint with each of its neighbors. Particles at the corners of the cloth will have be involved in three constraint relationships only, as they only have three neighbouring particles. Finding neighbors is easy using the grid based approach to the cloth construction as we can compare particles' x and y indices in the grid.

The final step is to make the middle particle unmovable. This anchors this particle and will allow the cloth to be positioned above the sphere in the Ghost class to allow the cloth to drape over the sphere to create a ghost-like object. More details are given in Section 2.3.7.

drawShaded()

This function sets up the normals of the particles properly and then calls the previously defined drawTriangle function while iterating over the grid to render all the triangles in the grid.

We begin by resetting the normals on all the particles by using the resetNormals function defined in Section 2.3.3. We then iterate through the grid and compute the normals for the triangles that encompass the current particle and its neighbors. The normal of each triangle is added to the accumulated normal of the particle using the addToNormal function defined in Section 2.3.3. This makes sure that the normal of all the triangles that that particle contributes to are involved in the definition of the particles normal and allows for smooth transitions between that particles and its neghbors when it comes to drawing triangles, since their normals have been factored in already.

timeStep()

This function is called once per frame and its main purpose is to make sure

that constraints are satisfied. To do this we iterate through the constraints vector container and satisfy each constraint many times. This allows for us to create a smooth looking motion by recomputing the correction vector many times. The number of times this is done is arbitrary, but the higher the iterations the smoother the results, but at higher computation cost.

Once all the constraints have been satisfied we call the timeStep function on each particle (defined in Section 2.3.3) in order to compute the new position of the particle before re-rendering it.

addForce(Vec3 direction)

This function adds a force to the cloth by adding that force to each of the particles one by one (calls the addForce function on each particle defined in Section 2.3.3).

windForce(Vec3 direction)

This function adds a force to each triangle in the cloth using the addWindForcesForTriangle function to apply wind to the entire cloth. The effect of the force depends on the dot product of the wind direction and the triangle's normal as explained previously.

ballCollision(Vec3 center, float radius)

This function determines if a collision between the cloth and the sphere underlying it has occurred and if so, resolves it.

The first step is to determine if a collision has occurred. We do this on a per particle basis. For each particle, we compute the distance from the particle to the center of the sphere (center in the function definition) by taking the length of the vector obtained by doing the vector difference of the two points.

We then compare this length to the radius of the sphere and if it is less, then this means that the particle has penetrated inside the sphere (i.e. a collision has occurred). If this is the case, the particle should be moved back to the surface of the sphere. This is done by first normalizing the vector connecting the particle and the center of the sphere and multiplying it by the radius of the sphere minus the length of it. This vector is then used to offset the position of the particle and effectively projects it back to the surface of the sphere where it would have collided.

move(Vec3 move_speed, Vec3 ball_pos)

This function is used to move the cloth as a whole. It iterates through each particle and if the particle is movable, it sets its position to the current position

plus the move_speed of the cloth.

setColor(Vec3 c)

This utility function just resets the color of the cloth to a Vec3 that contains an RGB representation of the desired color.

2.3.6 Game Board

This class represents the board on which the game is played, and defines where obstacles are placed as well as the tile position convention used in the Ghost class described in Section 2.3.7 as well as the pacman class. The code in this class comes from the base pacman game elements taken from [see ref 1] and can be found in the file gameboard.h, found in the file structure shown in Figure 2. It also contains a function and BITMAP for texture loading for the walls that can be used to apply custom textures to the board. This functionality was not used in our implementation.

The class contains many data members, all of which are explained below.

DIM_X A static, constant integer that holds the size of the game board in the x direction.

DIM_Y A static, constant integer that holds the size of the game board in the y direction.

CENTER_X A static, constant integer that holds the x position of the center of the game board. It is computed by simply dividing the size of the game board in the x direction by 2.

CENTER_Y A static, constant integer that holds the y position of the center of the game board. It is computed by adding 1 to the size of the game board in the y direction and then dividing that by 2. The reason for this is to omit the offset from the wall border of the board in this direction.

CENTER_Z A static, constant integer that holds the z position of the center of the game board. Its value is zero, since everything on the board lies in the same plane and all tiles on the board have the same z value.

coinsCount An integer that indicates how many coins are currently on the board. When pacman moves over a coin on the board, this value should be decremented. When it reaches 0, the player has won the game.

initial_map A two dimensional array of size $DIM\_X \times DIM\_Y$ that holds the map of the board. Each element contains an integer that indicates what should be rendered on that tile. The integers used for different game elements are enumerated below. There are actually more possibilities outlined in the comments of this class in the code, but for our purposes we only use the ones outlined below.

0 This indicates that a coin should appear at that tile on the gameboard.

1 This indicates that a wall should appear at that tile on the gameboard.

3 This indicates that an energizer should appear at that tile on the gameboard. When pacman moves over this, he will become powered up, allowing him to eat ghosts. This is described in some detail in Section 2.3.7.

BM This is a bitmap type object that can be used to apply textures to the gameboard using the TextureLoad function described below. This is not used in our implementation but was part of the code used as the base for our game found in [see ref 1].

The map of the game is instantiated in the gameboard.cpp file as shown in Figure 5. In the comments, we see the convention for the x and y tiles. That is as we move down, the y values increase and the x values increase as we move right. This is the convention used in the Ghost class but not the Pacman class which is why the adjustment described in Section 2.3.7 was used. The values here are the numbers described above.

Figure 5: Game Board Initialization

```
//0                  1         5         2                  29
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, // 0
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1}, // 1
 {1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1,1,1,1,0,1,0,1,1,1,0,0,0,1,0,1}, // 2
 {1,3,1,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,1,0,1,0,1,3,1},
 {1,0,1,0,1,1,1,1,0,1,1,1,1,1,0,0,0,0,0,1,0,1,1,1,1,0,0,0,1,0,1,0,1}, // 4

 {1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,1,1,0,0,1,0,1,0,1,0,1}, // 5
 {1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,1,0,1,1,0,1,0,1,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1},
 {1,0,1,1,1,1,0,1,1,1,0,1,1,0,1,0,1,0,0,0,0,0,0,1,0,1,1,0,1,1,1,1,1},
 {1,0,0,0,1,0,0,0,1,0,0,0,0,0,0,1,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1},

 {1,1,1,0,1,1,1,0,1,0,1,1,1,1,1,0,0,0,0,1,0,1,1,0,1,1,1,1,0,1,0,1}, // 10
 {1,3,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,3,1},
 {1,0,1,1,1,1,1,1,0,1,1,1,1,1,0,1,0,0,0,0,1,1,1,1,1,1,0,1,1,1,1,0,1}, // 12
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1}, // 13
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, // 14
```

The class contains many functions which are described in detail below.

bool isWall(int x, int y) A function that returns true if the tile at position (x, y) on the gameboard contains a wall, that is if the value at position (x,y) in the map shown in Figure 5 has the value 1.

GameBoard() A basic constructor that sets the coin count by iterating through the map and counting all of the coins

on the map. This constructor also loads textures if needed for the walls, using the function described below. However since we are not using textures on the walls in our implementation this has no effect.

void Draw() This function draws the game board in its entirety by calling all the other draw functions of the class. It also does the rotation animation on the coins. For each coin on the gameboard, every time this is called, the coin is rotated by a small step at the beginning of the code. Then each coin is drawn with that orientation using OpenGL functions to make it look as though the coins are rotating.

void DrawWall(int x, int y, int z) This function will draw a wall using OpenGL functions at the position indicated by x, y, z.

void DrawWalls(int j, int i) This function will draw a wall for each position that should have a wall in the game board map. It verifies the angle of the wall by checking the tile above or below and to the left or right to determine if those also have walls. For example, if tile (1, 1) has a wall and tile (1, 2) has a wall as well then the wall should be horizontal.

Once the angle has been decided, we need to determine if the wall has ended. For example if the wall is horizontal and the tile 2 tiles to the left or right of it is not a wall then the tile next to it should be a wall end. In that case we call the DrawWallEnd() function to draw a wall end. Otherwise we call the normal DrawWall function to draw a wall at that tile.

void TextureLoad(int id) This function is used to load textures that you would want to apply to the walls. This is not used in our implementation so it is not described here.

void DrawWallEnd(int x, int y, int z, int angle) This function draws a wall ending using OpenGL functions. It tapers off the wall and is called in the DrawWalls function when a specific condition is met as described above. It is not part of the GameBoard class.

2.3.7 Ghosts

The Ghost class contains two major parts. The first is a cloth, and the second a sphere. The cloth is anchored at its midpoint as defined in Section 2.3.5. It is then positioned above the sphere and centered to it, then allowed to fall and

drape over the sphere creating a ghost-like object. The cloth is essentially a child of the sphere and so by moving the sphere, we move the cloth as well, making the whole "ghost" move as one object.

The ghost.cpp file, which can be located by looking at Figure 2, also contains some globally defined constants that are worth mentioning. These are listed below.

TILE_SIZE_LEFT A float value that indicates the size of each tile on the gameboard when moving left. This is necessary to know when determining which tile the ghost is currently on. More details on how it is used follow this list.

TILE_SIZE_RIGHT A float value that indicates the size of each tile on the gameboard when moving right. This is necessary to know when determining which tile the ghost is currently on. More details on how it is used follow this list.

TILE_SIZE_UP A float value that indicates the size of each tile on the gameboard when moving up. This is necessary to know when determining which tile the ghost is currently on. More details on how it is used follow this list.

TILE_SIZE_DOWN A float value that indicates the size of each tile on the gameboard when moving up. This is necessary to know when determining which tile the ghost is currently on. More details on how it is used follow this list.

INTERSECTION_DIR_CHANGE_PROB An integer that indicates the probability that the ghost will change its course when moving past an intersection on the gameboard. This is important as it introduces some random motion to the ghost and makes its motion through the map and from playthrough to playthrough unpredictable.

TRACKER_RANGE_STOP An integer that indicates the range (in tiles) at which a ghost with AI_TYPE TRACKER (more details to follow) will stop tracking Pacman. This is important because if this range is set to 0, the ghost makes an instant beeline to Pacman making the game extremely difficult and unplayable. To remedy this, the range brings the ghost within a specified range of Pacman. While the ghost is within this range, its movement is dictated as if the AI_TYPE was set to NONE which is essentially random motion. This makes escape more likely and the ghost movement more unpredictable.

The ghost.h file which can be located by looking at Figure 2, defines the Ghost class but also defines some important enumerator types as shown in Figure 3. These are explained below.

The first is an enumerator that indicated the direction of motion of the ghost and is called MOVE_DIRECTION. Its values are explained below.

UP This value indicates a move direction in the positive y direction on the gameboard.

DOWN This value indicates a move direction in the negative y direction on the gameboard.

RIGHT This value indicates a move direction in the positive x direction on the gameboard.

LEFT This value indicates a move direction in the negative x direction on the gameboard.

The second is an enumerator called AI_TYPE that contains two values, that indicate different ghost movement types. The movement types for each AI_TYPE are outlined briefly here, but more details can be found in the move function description for the ghost (including implementation details).

NONE This AI_TYPE essentially represents random motion for the ghost. Each time the tile is updated, the ghost has a chance of updating his movement direction. Looking at Figure 6, we can see that the ghost is moving in the RIGHT direction. An intersection is indicated by a cyan colored box. The next tile the ghost will reach has a wall above and below it. Since it is moving right, these are the only possible course change options. However, since the tile above and below contain walls, a course change is determined to be impossible and the ghost keeps moving right.

The first scenario is the following:

The next tile in the direction of motion contains an impassable object, requiring a change in motion. The procedure for handling this case is outlined below.

Once the ghost reaches the intersection tile, the tiles above and below it are both available for the ghost to move toward. However, the tile to the right contains a wall. Therefore a course change in the vertical direction is necessary. We first check the tile above and find that is is free and so change course by setting the MOVE_DIRECTION for the ghost to UP. If there had been a wall on the tile above, the movement would have changed for the ghost to go down.

A different scenario occurs when the ghost is moving through an intersection where the next tile does not contain an impassable object. The procedure for handling this is outlined below.

Assume the ghost in Figure 6 is moving left instead of right. The first intersection it reaches contains a wall below, none above, and also none to the left. At an intersection like this, a course change is not required since the ghost could keep moving left. However, to make the movement more random, when this situation occurs, a random number is generated and compared to the INTERSECTION_DIR_CHANGE_PROB defined previously. If the number is less than this probability, a course change takes place. Otherwise the ghost keeps moving left. If the course change were to take place in the example, the ghost would begin moving up since this is the only valid course change due to the wall in the tile below the intersection tile.

TRACKER This AI_TYPE makes the ghost attempt to follow pacman. Each time an intersection is reached, a course change is tested for (if required to follow pacman) and imposed on the ghosts movement direction. An example is shown in Figure 7.

Here the ghost (blue) is moving to the right. Once an intersection is reached (the first one being the one with a breakable wall indicated by the red cube), a test is done, comparing the tile positions of the ghost and pacman. We determine that pacman is below us, however, a wall is disallowing a course change down. Next we determine that pacman is to our right and so we "change" our course to move right, moving us closer to pacman. This is only done if the difference between the ghost's tile position and pacman's position is greater than the threshold for TRACKER to be enabled (TRACKER_RANGE_STOP) in the appropriate dimension. If it is not, then the ghost "reverts" to the NONE AI_TYPE and performs course changes at intersections in pseudo-random fashion until pacman has moved far enough away to be tracked again.

The data members of the class are explained in detail below.

ball_pos A Vec3 that indicates the current position of the sphere which is also the way we define the position of the ghost (i.e. the ghost's position IS the position of the sphere).

ball_radius A float that indicates the radius of the sphere.

cloth A Cloth object as defined in Section 2.3.5, that is draped over the sphere to create the ghost appearance. This cloth is a child of the sphere and so its movement direction is dictated by the movement of the sphere to ensure the composite parts of the ghost move as one.

Figure 6: Ghost Movement Algorithms (NONE)

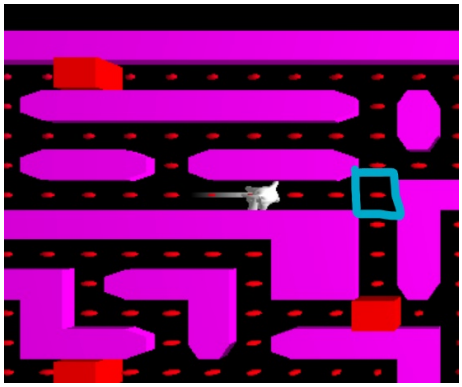
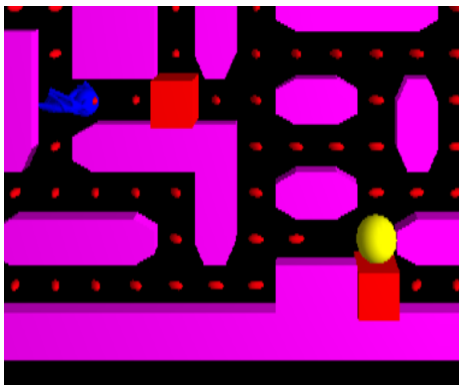


Figure 7: Ghost Movement Algorithms (TRACKER)



move_speed A float that indicates the speed of motion of the ghost. When the move function is called, the ghost is translated by the move_direction vector scaled with the move_speed modifier. This separates the direction of motion from the magnitude of the motion.

move_direction A unit Vec3 that indicates the direction of motion of the ghost. For example, if the ghost's MOVE_DIRECTION is UP, this field will be the Vec3 (0, 1, 0).

originalClothColor A Vec3 that stores the original cloth color in RGB format. This is necessary since when Pacman picks up a powerup, the ghost's color will change to cyan indicating they can be eaten. When the powerup expires, the color will be reset so it must be stored somewhere.

color A Vec3 used to define the color of the ghost in RGB format. Changing

this will change both the color of the cloth and the sphere to the specified color.

ghost_pos A Vec3 used to track the position of the ghost at any point in time. The position of the ghost at any time is equal to the position of the sphere.

tile_x An integer that indicates the x value of the tile the ghost is currently on. The tiles with an x value of zero are the tiles containing the left-most wall (the left border) of the gameboard.

tile_y An integer that indicates the y value of the tile the ghost is currently on. The tiles with a y value of zero are the tiles containing the top-most wall (the top border) of the gameboard.The y values of tiles increase down the gameboard.

acc_move A float value that increases as the ghost moves between tiles. This is used to indicate if the ghost has reached a new tile or not. For example let's assume the ghost is moving to the right. As this is happening, the x value of its position is increasing by its move_speed every time it moves. Thus every time the move function is called on the ghost we increment acc_move by an amout equal to move_speed. When acc_move reaches the tile width in the direction of motion (in this case it would be TILE_SIZE_RIGHT), we increment the appropriate tile value, in this case we would increment tile_x and then reset acc_move to 0 since we are now on a new tile.

dir A MOVE_DIRECTION type that indicates the direction of motion of the ghost. If the ghost is moving right, this will have the value RIGHT.

pacman This is a pointer to a Pac object, in this case it is a reference to the pacman object in the game. It is necessary to be set for the AI_TYPE TRACKER to work since it tries to match the position of the ghost to the position of pacman.

ai_type An AI_TYPE type that indicates which type of AI the ghost is using to dicate its movements. The various types were explained in detail previously. Implementation details to follow in the functional descriptions.

The functions of the class are explained in detail below.

bool isWall(int x, int y)

This function returns true if the tile at position (x, y) in the gameboard contains a wall and false otherwise. It uses a function defined in the GameBoard class. The details of what consists of a wall on the gameboard are explained in Section 2.3.6.

bool isIntersection(int x, int y)

This function determines if the position (x, y) on the game board is an intersection or not. This is used to check if a course change for the ghost is possible or not. If it is not, then course change checks as described previously are simply skipped.

For example, if the ghost is currently moving either right or left, an "intersection", i.e. a potential course change opportunity, exists if either the tile above or below does not contain a wall. If both the tile above and below the current tile have walls then this is not an intersection and the function will return false.

bool ai_move()

This function moves the ghost based on its AI_TYPE and is used in conjunction with the move() function of this class. As described previously, there are two AI types available, NONE and TRACKER. Upon execution of this function, if the ghost's AI_TYPE is NONE, then we return from the function since its movement is not governed by AI but rather random as explained previously.

If this AI_TYPE is TRACKER, the ghost attempts to move towards pacman. Therefore we need to know the current position of pacman. Since in the original code, pacman's current tile is defined in a coordinate convention opposite to what is indicated in the gameboard by the author of [see ref 1] for the y direction, an adjustment of pacman's current y tile position is necessary. This is done with the formuila shown in Equation 9, which maps pacman's y tile coordinate to the system used by the ghosts and the gameboard class. Once again, details of this system are indicated in Section 2.3.6.

$$pac\_y = GameBoard :: DIM\_Y - pacman- > tileY - 1 \tag{9}$$

Once we have pacman's tile position, we try to move the ghost towards it. It is also important to remember that this is only done if the ghost is outside the TRACKER_RANGE_STOP range as explained previously. We first check this, and if this check passes, we then verify if pacman is to the left of our ghost by comparing their x tile positions. Essentially if the ghost is at an intersection (meaning a course change is possible) and its x tile is greater than pacman's x tile, the we change its move direction to LEFT if possible (in the case where the tile to the left of our ghost is not a wall) and return true (indicating that a course change was made). If the tile to the left of our ghost contained a wall, then we return false indicating that no course change was made. The reason we do this is to ensure that no further course changes are allowed at this intersection since ai_move is called before regular move checks are made in the move function. If we return true, then we skip normal movement. More details are given in the description of the move function.

If pacman was not to the left, but instead to the right, i.e. the tile x value of the ghost was less than the tile x value of pacman, then we attempt to move the ghost right and return true or false based on the result in the same fashion. The same procedure is done with the above and below checks, the difference being that we use pacman's mapped tile y position to do the check fot the reason explained previously.

Ghost(float scale_factor, Vec3 color, int tile_x, int tile_y)

This function while spawn a ghost at the tile on the gameboard indicated by tile_x and tile_y colored according to the RGB value specified by the Vec3 color and with the appropriate size indicated by the scale_factor parameter.

By default the AI_TYPE for the ghost is set to NONE.

Many careful adjustments are made in the constructor to ensure that the ghost is spawned at the appropriate depth in world space to appear as if it is in plane with the gameboard. Also, the srand function is used to seed a random number generator based on the current time to ensure that the random numbers generated to determine if course changes should occur at intersections are different each time the game is played.

void display()

This function does the actual frame by frame rendering of our ghost object. First, the ghost is moved by calling the move function explained below. Next, a gravity force is applied by using the addForce method of the cloth explained in Section 2.3.5. This gravity force points "down" which in this case means into the gameboard i.e. the negative z direction. This force is proportional to the speed of the ghost and ensures that the cloth stays draped over the ball and does not fall off at higher speeds

Next, a wind force is added to the cloth, also proportional to the speed in the direction opposite the ghost's move direction to make the cloth appear as though it is trailing behind the ghost as it moves.

Next the cloth timestep function described in Section 2.3.5 is called to recompute the particle positions for the cloth based on the added forces. Finally we check for collisions between the cloth and ball as described in Section 2.3.5 and draw the cloth using its draw function.

The ball is then drawn using the glutSolidSphere function (part of the GLUT library) at the position indicated by the ball_pos data member of the class.

void move()

This is the function called to move the ghost and is the most important function in the class, as well as the most complex. We begin by moving the ball, by changing ball_pos by a factor indicated by the speed data member of the class in the direction indicated by move_direction. We then move the cloth using its move function described in Section 2.3.5. Next, acc_move is incremented by a factor equal to the speed of motion. This function also has a boolean that indicates if the direction was changed already during the execution of the function. Only one direction change is allowed per execution. If this is not enforced, then say for example that we check if pacman can move up at an intersection first, and check if he can go down only after this check is done. If pacman arrives at an intersection where he can move BOTH up and down, the check to go up passes, and his direction is changed to UP. However, when checking if pacman can go down, the answer is also yes, and his direction will change to go DOWN. This means that at this intersection, it is impossible for pacman to ever go up. To prevent this, we introduce this boolean to make sure we can only change direction once.

Now we begin by checking if acc_move is greater than or equal to the tile size in the direction of motion as explained previously. If it is, then we are at a new tile and the direction could potentially change. When this occurs, we reset acc_move to 0 to indicate we have reached a new tile and increment the tile_x and tile_y for the ghost according to the direction of motion.

Assume that the ghost is moving up, and we have not yet changed the direction, i.e. our boolean for this is still false. The first thing we need to check is if a course change is REQUIRED. This occurs (in this situation) if the tile above the one we are on is a wall (the tile at our tile_y - 1 in our convention). This means we cannot continue moving up and we must change course to either left or right. To determine which one it should be we check if the tile to the right is a wall (the tile at our tile_x - 1) using the isWall function for that tile. If it is, then we change the MOVE_DIRECTION to RIGHT using the appropriate setMoveDirection function described below and set our direction changed flag to true to prevent further course modifications in this frame. If the tile to the right is a wall, then we change the MOVE_DIRECTION to LEFT in similar fashion and set the direction change flag to true. A similar procedure is done for other REQUIRED course changes but with different input MOVE_DIRECTION.

Once these forced course changes are dealt with, we attempt a course change based on the ghost's AI using the previously explained ai_move function. If this function returns true, it indicates that the course was changed by the ai and no further course changes should be allowed so as to not destroy the decision taken by the AI model. This is ensured by returning from the function if ai_move returns true.

If ai_move returns false, i.e. no course change was made by the AI model, then we try to move the ghost in a random way as described previously. To do

this, we check if the ghost is at an intersection using the isIntersection function described previously for the current tile and that the direction has not already been modified by a REQUIRED course change as we do not want to undo this. If these checks pass, a random course change is allowed.

To determine if a random course change should occur, we generate a random number between 1 and 100 using the standard rand() function. If the number generated is less than INTERSECTION_DIR_CHANGE_PROB, then a course change can happen. Let's explain with an example.

If a course change can occur and we assume pacman is moving UP, then possible course changes are to LEFT or RIGHT. We first check the RIGHT direction. If the tile at position (tile_x + 1, tile_y) is not a wall (checked with the isWall function), and the random number we generated is larger than 50, the we change course to RIGHT. otherwise if there is no wall to the LEFT and the random number generated was less than 50 we go LEFT. This model makes sure there is an equal chance of going RIGHT or LEFT in this case (assuming the course change is possible, i.e. no obstacle in the desired new direction of motion). Note that it is possible for no course change is also possible even if our random number was less than INTERSECTION_DIR_CHANGE_PROB. This can occur if say there is no wall to the right, but the random number generated was 30, which means that the course change should be to the left, but there is a wall to the left preventing course change in that direction. This introduces even more unpredictability to the motion. A similar method is used for an initial move direction in other directions, i.e. DOWN, LEFT, or RIGHT.

void setMoveSpeed(float speed)

This function is used to set the move_speed of the ghost to a new value. This move_speed is multiplied by the move_direction unit vector as described previously when moving the ghost to scale movement in the desrired direction.

void setMoveDirection(MOVE_DIRECTION dir)

This function changes the move direction of our ghost based on the MOVE_DIRECTION type passed in. This function needs to change two things: the MOVE_DIRECTION type of this class and the move_direction data member of the class which is a unit Vec3 used for doing actual computations when moving the ghost. For example, if the value UP is passed in to the function, the dir data member will be set to UP, and the move_direction data member will be set to a new Vec3 with the values (0, 1, 0).

void setColor(Vec3 color)

This function sets the color of the ghost to the Vec3 passed in, which contains the new color in RGB format. This will change the color of the cloth and

the ball.

void setGhostColor(Vec3 color)

This function sets the color of only the cloth to the Vec3 passed in, which contains the desired color in RGB format. This will change the color of the cloth while leaving the ball unaffected. This is used when changing the ghost's color to cyan when a powerup has been picked up by pacman as explained previously to change only the cloth color.

void setOriginalGhostColor()

This function resets the color of the ghost to its original color. This is called when the powerup that pacman picked up expires. Since the ghost can no longer be eaten, its color changes back to whatever was set originally. So if the ghost was initially green, this will change the color from the cyan color used when the ghost can be eaten back to green.

Vec3 getPosition()

This function returns a Vec3 containing the x, y, and z positions of the ghost in WORLD SPACE. This Vec3 coincides with the ball_pos data member of the class, which is also the position of the center of the sphere in WORLD SPACE. This is how the ghost's position is defined as specified previously.

void setPacman(Pac* p)

This function sets the class pointer to pacman to the pacman pointer passed in. This should be called in the main function, as the Ghost constructor does not initialize this. If it is not set, the ghost's with AI_TYPE TRACKER will not be able to follow pacman properly, as they need to know it's position in order to perform the required course changes to follow him.

void setAIType(AI_TYPE t)

This function changes the AI_TYPE for the ghost to the specified value. The possible values are NONE and TRACKER as explained previously which have different effects in the ai_move() function described above.

int getTileX()

Returns an integer containing the current x position of the ghost on the gameboard, i.e. the x value of the tile that the ghost is currently on. If a ghost is currently in between two tiles, the value returned is the x value of the last tile that the ghost was on. This value follows from the convention explained in

the Game Board class, which is explained in Section 2.3.6.

int getTileY()

Returns an integer containing the current y position of the ghost on the gameboard, i.e. the y value of the tile that the ghost is currently on. If a ghost is currently in between two tiles, the value returned is the y value of the last tile that the ghost was on. This value follows from the convention explained in the Game Board class, which is explained in Section 2.3.6.

Vec3 getBoardPosition()

This function gets the position on the gameboard not in tiles but in true (x, y, z) fashion. This is different from the getTileX and getTileY functions explained above since those return integers. This function returns a Vec3 that will say, for example, if the ghost is between the tiles (2, 3, 0) and (3, 3, 0), that the board position is (2.42, 3.0, 0.0).

Let's assume that the current direction is UP. The the vector will contain the current tile_x that the ghost is on in it's x position. However, the y position may be in between two tiles. To figure out where it is we use Equation 10. Here we subtract the ratio of the current acc_move and the tile size in the direction of motion from the current y tile since the y value for the tile decreases as we go up as per the convention established in Section 2.3.6. This will give us a number between the current y tile and the next, which yields a Vec3 similar to what was shown above. The procedure is similar for other move directions.

$$boardPosition_y = tile\_y - \frac{acc\_move}{TILE\_SIZE\_UP} \tag{10}$$

void collision(Pac* pac)

This function is used to determine if a collision has occurred between the ghost and pacman. This is done by checking if the x and y tile positions of the ghost and pacman on the grid of the gameboard are matching. If this is the case, we move to collision resolution, otherwise we do nothing.

To resolve the collision, we need to check if pacman is powered up first, in which case the ghost should be eaten, i.e. be removed from the game. This is not implemented yet and could be a further addition to the game, as described in Section 3.

If pacman is not powered up and not invulnerable, then pacman should lose a life, done by decrementing the lives data member of that class and making him invulnerable for a time so he doesn't lose more lives while still colliding. A

sound is also played using the Sound Manager class described in Section 2.3.10. If pacman has run out of lives, a particle effect is generated to show that pacman has died using the TrailParticles class described in section 2.3.13.

2.3.8 Pacman

The pacman class, defined in Pac.h found in the File Structure shown in Figure 2, is where all of pacman's properties are set such as speed, angle, lives and certain booleans for when pacman is dead or is powered up. These properties are all set in the constructor.
To make pacman move properly throughout the map, the tile position of pacman is needed. This is receieved through the Pad() function which takes the world coordinates of pacman which are floats and casts them to an integer so that pacman is always centered in the tiles.

The function WallCheck() returns a boolean to see if pacman can go towards the next tile he is facing. This prevents pacman from going into a wall.

The Turn(int angle) function which is called in the keyboard function in main takes pacman's current angle and will turn pacman depending on the button that was pressed and if pacman can actually turn in the given direction.

The Move() function calculates the x and y position increments it by a certain value every frame as long as pacman is able to move.

2.3.9 Main.cpp

Main.cpp, which can be in the File Structure shown in Figure 2, consists of an initializer to initialize all the objects and their properties the game needs such as ghosts, pacman, the gameboard, etc. It consists of the display function which contains the main game loop which processes all the game logic. It also includes the main function which is used to initialize glut and register keyboard and mous callback functions.

Globally, all the objects that will be needed throughout the main file are created. This includes the ghosts, particle systems, the breakable walls and the SoundManager instance.

The init() function sets the properties of each ghost such as color, move direction and AI type. All the game sounds are also loaded.

In the display() function is where the main game loop takes place and all game logic is processed. This processes the pacman and ghost movement. It checks for collisions between pacman and the ghosts. The necessary particle effects are also processed in this loop. Just like a game loop in any game, the

drawing step is done last.

The keyboard(unsigned char key, int x, int y), mouse(int button, int state, int x, int y) and special(int key, int x, int y) functions are where input is processed. The keyboard and special functions both process keyboard input but GLUT processes special keys such as the arrow keys and character keys differently which is why two functions are needed.

In the (int argc, char** argv) function, all GLUT properties are initialized. The pacman and board objects are initialized and the glutMainLoop() function is called. The function to launch the kinect is also called here.

2.3.10 Audio

To achieve the use of audio, we attached the SFML library to the project.

The file SoundManager.h is the singleton class that implements the functions to load and play sounds. While we are aware of the general dangers of singleton, we chose to make it one for its speed and ease of use. To use it, a file simply needs to include SoundManager.h.

With SFML, each sound file needs a sound buffer (sf::SoundBuffer) and a sound object (sf::Sound) to play correctly. There is an enum called SoundEffects that has a value for each sound effect in the game.

There are two dictionaries in the class; one for buffers (SoundEffects, sf::SoundBuffer) and one for sounds (SoundEffects, sf::Sound).

The function loadSound takes as parameters a SoundEffect enum value and a filename for the audio file that needs to be loaded. This function creates an SFML sound buffer and sound object and calls the SFML loadFromFile function and places the sound buffer and sound object into their respective dictionaries where the key is the SoundEffect parameter.

The function playSound takes a SoundEffect enum parameter and simply accesses the sound it needs by the key value and calls the SFML play() function.

An example usage of the SoundManager singleton would be as follows:

SoundManager::Instance() $\rightarrow$
$loadSound(SoundEffects :: PacmanChomp, "pacmanChomp.wav")$;

SoundManager::Instance() $\rightarrow Play(SoundEffects :: PacmanChomp)$;

2.3.11 Delta Time

Every game needs some kind of delta time functionality which is the time difference between the previous frame and the current frame. This allows games to add time based mechanics.

Since OpenGL does not have a built in delta time functionality, we created our own.

The Time.h class simply has a static float variable called deltaTime. The deltaTime is calculcated at the beginning of the main loop. Any class that needs deltaTime to simply needs to include "Time.h" and statically access the variable.

2.3.12 Pacman Collision

Pacman collision includes how collision is done with the ghosts as well as the coins and power ups.

The game board is made into tiles and each object in the game has a tile position as well as a world position. The tile positions of the objects are used to check if there is a collision between objects or not.

To check for collisions, we simply check for when the x and y tile positions of pacman are equal to the x and y tile positions of ghosts/coins/power ups and if they are, then the game logic happens. To see an example of this, look at the collision function implemented in ghost.cpp.

2.3.13 Trail Particles

The TrailParticle class describes the particle effect seen on pacman and the ghosts as they move through the gameboard. This class is also used when the explosion effect is triggered the moment pacman dies or pacman grabs one of the energizers.

The TrailParticle class has many data members, whose functions are enumerated as follows:

location It's a Vec3 that describes the location of a particle at particular point in time.

velocity It's a Vec3 that describes how fast and in what direction the particles is moving at.

acceleration It's a Vec3 that is added to the velocity in order to replicate acceleration over time.

color It's a Vec3 that describes the color of the particle

alpha It's a float that describes the transparency (How visible it is) of the particle. It's range is 0-1.

lifespan It's a float which describes the amount of time the particle is going to remain "alive".

activateEnergizerParticles It's a static boolean that triggers the explosion effect when pacman gets the energizers.

The TrailParticle class has functions that allow for the updating of a particles location, velocity, and displaying.

TrailParticle()

A default constructor.

TrailParticle(Vec3 location, Vec3 velocity, Vec3 acceleration, Vec3 color, float alpha, float lifespan)

A constructor that takes in all the basic requirements of a particles (Location, Velocity, Acceleration, Color, Alpha, Lifespan) and sets them the passed value.

update()

A function that updates the particles location and lifespan/alpha by adding the acceleration vector to the velocity vector and then adding the velocity vector to the location vector, while diminishing the lifespan and alpha values.

display()

A function that takes care of the "drawing" of particles on screen. In this case the "drawing" used is a simple glutSolidCube with the corresponding color and alpha values passed on through the constructor.

isDead()

A function that returns a boolean telling us wether the particle is dead or not. (If lifespan or alpha = 0).

setEnergizerParticles(bool set)
A static function that can set the value of the activateEnergizerParticles static boolean data member.

isEnergizerPSActivated()

A static function that returns a boolean checking to see if the EnergizerParticles are currently activated.

2.3.14 Particle System

The purpose of the ParticleSystem class is to serve as a container for all the instances of a particle.

The ParticleSystem class only has one data member:

particles It's a C++ vector (Dynamic Array) that contains particles of type TrailParticle

The ParticleSystem class basic functions that initialize, add and update the particles.

addPartice(TrailParticle p) A default constructor.

run() A function that runs through the particles array and calls each particles update and display methods. It also updates the size of the array by removing any dead particles.

getSize() A function that returns the current size of the particles array.

2.3.15 Breakable Walls

Breakable Walls Class

Breakable walls in the game are done using a simple particle system.

The classes used are as follows:

1. BreakableWall.h
2. BreakableWall.cpp

These can be found by looking at the File Structure shown in Figure 2.

This class depends upon GLUT for drawing the wall and its particles as well as manipulating translation and rotation for the particles.

The class also includes Creature.h found in the File Structure shown in Figure 2, which is a parent of the Pac class.

This is required to find the collision between pacman and the walls of the gameboard explained in Section 2.3.6, upon collision, the particle system activates.

Class Format

BreakableWall has a constructor that takes in 4 parameters:

BreakableWall(float posX, float posY, float posZ, float wallSize)

Where x,y,z positions are given, along with the size of the wall. Since it is a top down view of the game, it is better to have the z set to 1, and the x, y positions to be changed according to where the walls are required.

To fit the walls exactly on the tiles on the map, the values passed in for x and y should be integers. Recommended wall size is 1 unit, which fits a tile completely.

Wall objects are kept in an array in main.cpp found in the File Structure shown in Figure 2, and in the main loop, the draw function is continuously called while iterating on these objects.
Before calling the draw function, we check if the particle system has expired, which is a boolean in the class. If this is true, the main.cpp will not call the draw function on this object any longer.

It has a function to check pacman's position. If they are on the same tile as the wall, the wall is destroyed.

2.4 Use Cases

2.4.1 Scenario 1

Pacman runs into a wall.

⇒ Check if the tile Pacman will go to next is empty if not a collision with a wall is detected

⇒ When collision is detected stop Pacman from further going in that direction

2.4.2 Scenario 2

Pacman runs into a pickable (coin or cherry).

⇒ Check if the tile Pacman is on and the pickable tile are the same

⇒ if so the pickable disappears and the corresponding effect occurs (lowers total coin count for coin or give powerup for cherry)

2.4.3 Scenario 3

Pacman runs into a ghost.

⇒ Check if the tile Pacman is on and the ghost tile are the same

⇒ if so the pacman loses a life

2.4.4 Scenario 4

Pacman runs into a breakable wall

⇒ Check if the tile Pacman is on and the breakable wall tile are the same

⇒ if so the wall disappears and leaves behind a particle effect=

2.4.5 Scenario 5

Keyboard input test

⇒ Check each keyboard input

⇒ Pacman should move accordingly in the direction of the input pressed as denoted in the user manual

2.4.6 Scenario 6

Kinect input test

⇒ Check each Kinect input

⇒ Pacman should move accordingly in the direction of the input done as denoted in the user manual

3 Conclusion

3.1 Limitations and Challenges

3.1.1 Performance

Performance is affected due to three reasons in this program:

Ghosts cloth simulation
4 cloth simulations running simultaneously slows down the program. If the program is run on a laptop, without being plugged in, then the Frames per second will be visibly low.

There is no frames counter implemented in this project as of now.
But could be added into calculated exact drop in performance for every ghost added.

Ghost cloth simulation as of now runs on a single thread, and this performance can be boosted by running the cloth simulations on multiple threads in the future.

Kinect input stream
Kinect input processing for the game controls can be turned off using the key 'K' which will toggle controls between keyboard and keyboard & kinect.

However even if kinect input is toggled off in the program, as long as kinect is plugged in, it will still be receiving input and slow down the program.Unplugging the kinect will stop this slow down.

Kinect Camera display
If a user chooses to enable the kinect display camera to see themselves while playing the game, the program currently displays image using a point cloud that includes depth which is not necessary or of any use for the user.

All kinect pixels are rendered as points with depth, which is a 3D points depth cloud.

Performance could be massively improved, if this is replaced with a 2D render to texture since the user only will need to see themselves, and don't need the depth data.

3.1.2 Camera Controls

Zooming in the map is another limitation, as the reference project that was used, does not move the camera on zoom in, but instead scales all other objects

to a certain scale multiplier.

The original zoom function was removed since this will not work on the newly added ghosts.

As a result, there are currently no camera movement functions available. Functions to do so can be added into the main, that will directly manipulate the location of the camera. This would require testing in order to make sure that it does not bug out any other feature.

3.1.3 End game

As of now, if a user finishes the game, by collecting all coins on the map, nothing happens. The player can keep moving around.

This is due to the fact that there is only one map, but if there were more maps, they can be loaded in after finishing the current one (See scalability section for addition of more maps).

This is also true for losing the game. If pacman loses all 3 lives, the game does not end. Pac man will not respawn, but instead the game will keep running the ghosts around the map.

3.1.4 Overlap grid collision

Since collisions between pacman and ghosts are done using a grid system, these two spheres that represent pacman and the ghosts will have to overlap in order to actually collide. This collision detection can be improved upon.

This can be changed to an implementation of a bound sphere on both pac man and the ghosts. This can give accurate collision results by comparing their radius.Collision will happen exactly when they touch.

3.1.5 PowerUp

The power up are available in the game , when pacman goes over a ring he is able to consume it activating the power up, this changes the ghost color to light cyan. this however only give immunity to ghost and does not allow pacman to eat ghosts.

for this to be added we would need to add code to the ghost to have him be deleted if he was consumed and to have him respawn after a variable of time.

3.2 Scalability

3.2.1 More maps

The Pacman game can be expanded by creating new levels. Levels or the game board is created using the gameboard.cpp

Currently gameboard has only 1 initial map setting, which is a 2 Dimensional array, with X, Y tile positions, defining if there is a wall at every tile, by putting that specific value as 1. These walls (with value 1) will be drawn.

This can be changed to support having multiple levels, by reading text files that already have predefined levels in them. These can be loaded in the beginning by asking the user which level they want to play, or can be loaded as a next level when the current level is completed.

3.2.2 More ghosts

Currently ghosts are created as single objects, each defined as a certain line in the main. These ghosts can instead be created in an array, and controlled using iteration to support more ghosts on a map.

Additional Ghosts would slow down the program since all cloth simulation takes place on a single thread. This is a limitation and performance can be improved using multi-threading in the future.

3.2.3 Additional Controls

Alternative controls other than keyboard or kinect can be added such as Microsoft's Xbox controller, Steam controller, etc.

Given that the required SDKs and drivers are available on a system, their callback functions can be used to map these alternative controls.

To make use of the program more convenient, default controls can be assigned to the keyboard, and special key bindings can allow users to switch between either the Xbox controller or the kinect.

Also a more standardized way to get new input can be put into action so no matter where the input comes from it can act accordingly making it more scalable with the number of possible input from different peripherals (such as kinect , keyboard , controller etc..)

4 References

[1] Michal Daniel Dobrzanski. *Pacman Base Code*. URL: https://github.com/MichalDanielDobrzanski/Pacman3D.

[2] Jesper Mosegaard. *Cloth Simulation Tutorial*. URL: https://viscomp.alexandra.dk/?p=147.

[3] The Coding Train. *Coding Challenge: Simple Particle System*. URL: https://www.youtube.com/watch?v=UcdigVaIYAk.

[4] Serguei Mokhov. *Comp 477 - f19 - notes*. URL: https://users.encs.concordia.ca/~c477_2/lectures/notes.pdf.

[5] Microsoft. *Microsoft Kinect SDK v2.0*. URL: https://www.microsoft.com/en-ca/download/details.aspx?id=44561.

[6] Edward Zhang. *Kinect Tutorial*. URL: https://homes.cs.washington.edu/~edzhang/tutorials/kinect2/kinect0.html.