Three-Dimensional Pac-Man

Written by

Bradley Gardner

For

CS-485: Advanced Game Development

Giuseppe Turini

*Game Concept*

The concept, in its entirety is simple, to bring [the beloved Pac-Man](https://www.youtube.com/watch?v=dScq4P5gn4A) franchise out of the two-dimensional world and into the three-dimensional one. In Pac-Man, the player takes on the role of the titular character, collecting pellets in a maze whilst avoiding a collection of four ghosts who will kill the player; however, the player will also have access to power-ups that will allow him/her to defeat the ghosts for a short period of time. The player’s main goal is to collect all of the pellets in a level so that he/she may continue onto the next maze and obtain a high score. Both the two-dimensional and three-dimensional game will be played from a third person perspective so that the player can keep a constant overview of the map and the enemy ghosts.

Regarding the core mechanics of the game, the first mechanic would be that of the enemies. The ghosts are programmed to traverse the maze as well, searching for Pac-Man as opposed to collecting pellets. These ghosts follow different movement patterns in order to attempt to collide with Pac-Man. Initially, a collision between Pac-Man and a ghost will result in the player losing a life and being moved back to the start position, level progress will not be affected by this if the player still has at least one life remaining. If the player, however, is currently under the effects of a power-up, the collision will instead result in the ghost being defeated and needing to return to the “base” located at the center of the maze, as seen in Figure 1, where the ghost will respawn after a brief waiting period. If the player, however, does run out of life, the game ends and progress is reset.

In terms of progress and scoring, the game has three basic ways of earning points: collecting pellets, defeating ghosts and obtaining power-ups. The pellets, as they are meant to be most common items to appear within the maze, will be worth the least amount of points. The player needs only to steer Pac-Man into the pellet to collect it, no additional inputs are required. Aside from pellets, the player can also collect the power-ups on the map, originally denoted by large circles as seen in Figure 1, to earn points as well as defeating ghosts while under the effects of said power-ups. In addition, when the player earns a certain amount of points, 10,000 in the original game, the player earns another life in order to reward him/her for playing the game well. Progression occurs when the player collects all of the pellets and power-ups on the map, after collecting all that there is to collect in the maze, the player will be taken to a new maze where they must attempt to collect everything yet again.

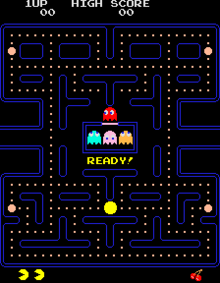


Figure 1: A two-dimensional Pac-Man maze, this displays the pellets used to score points as well as the power ups, depicted by the large circles on the map (Pac-Man Fandom, 2020).

Three-dimensional Pac-Man seeks to take those features of the original game and modify them for the three-dimensional world. While basic gameplay elements will remain the same, the game will now feature three-dimensional models as opposed to two-dimensional sprites as well as a three-dimensional maze for Pac-Man to navigate through. This allows for the game to bring a new spin on the classic Pac-Man game, while simultaneously keeping the experience similar to what people already know and love.

*Technical Features*

* Game Logic: A points system to provide the player with feedback. Points can be earned through the player collecting pellets, defeating ghosts or acquiring power-ups, with a point value being assigned to each based on the difficulty of the task. The game ends when the player runs out of lives
* State based character models for enemies to provide the player with feedback, show that attacks have successfully hit.
* A collision-based system to allow the player to collect points as well as be “killed” by enemies
* Running score counter present at the top of the screen for the player to keep track of their progress. The score counter is updated every time the player collects points.
* Life counter located at the bottom of the screen to keep the player up to date on their current number of remaining lives. Updates every time the player loses or gains a life.
* Power ups that reverse the collision detection of enemies and the player, allowing the player to kill enemies and score points.
* Third-person omniscient camera that the player can move forward and backward to get a better look at the maze.
* Sound will be comprised of the original, two-dimensional audio from the 1980 version.
* Player will always be able to view the entire map.
* Use of keyboard to obtain player inputs.
* Cartoon visuals that hold the spirit of the original game
* Tile based maze utilizing the A\* algorithm for path finding
* Optional: Animated 3D models.

*Main Game Mechanics*

In this game, the player’s take on the role of Pac-Man, as seen in Figure 2, and must navigate a maze whilst avoiding a collection of enemies. The player will collect points through a variety of different means including collecting point pellets and power ups, as well as by defeating enemies, as seen in Figure 2. Once the player has collected all of the points on a level, the player will be brought to the next level. In addition, the player will have a set amount of lives, four to start, and may acquire more by earning more points within the game. If the player runs out of lives, the game ends and it will be restarted.

The player will control the character with the keyboard, using the appropriate keys to go forward, left, right and backwards. The game will utilize a tile-based maze, featuring the A\* algorithm, to restrict movement to the tiles of the maze. In game enemies will also utilize this tile-based system in their path finding programing. In addition, the player will be able to use the front and back arrow keys to adjust the game’s camera, allowing for him/her to better view the upper sections of the maze.

The UI for the game is simple as to prevent from distracting the player, providing a basic overlay to display point and life information to the player. This will constantly update to keep the player informed as to what is happening in the game and to provide instance feedback based on their playing.

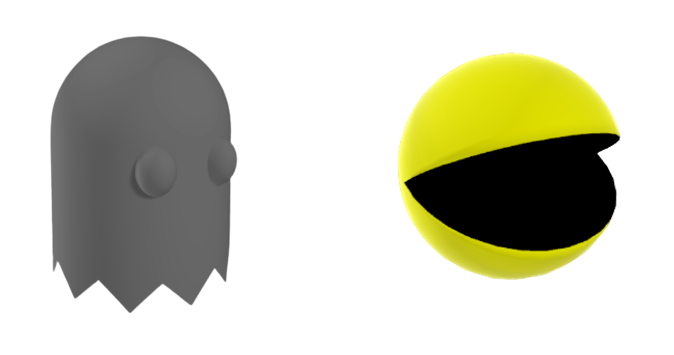


Figure 2: 3D Models of the generic ghost and Pac-Man

*Tile-Based Maze*

For movement, both enemy and player, follow a collection of tiles that make up the game’s maze. These tiles form a grid, constructed by the code as seen in Script 1, which denote the spaces that the player and enemies are allowed to move to, which is run by an invisible, empty object located within the maze. On the map are two, invisible blocks that are placed in the top right and bottom left corners of the maze, these are used to determine the height and the width by finding the difference between their x and z coordinates. Given this, it then determines the number of individual cells that exist within the world based on the size of a cell. This is done by finding the x and z distance between the upper right and lower left cells, this is then divided by the size of the cell, which in this instance is 1, to determine the total number of cells in the x and z directions of the grid.

After the generation of the grid, the game determines tiles that are not accessible. In the grid, all tiles that contain a wall are meant to be inaccessible, as the player, or an enemy, should not be able to simply walk through a solid wall and as such are fit with colliders as well as a layer mask named unwalkable. Once the grid has been created, the script then checks each tile within the grid to see if it is unwalkable. This is done by looping through all the cells within the two-dimensional array that constitutes the grid with a small sphere of with radius 0.4f, ensuring that it does not go beyond the boundaries of the tile. This sphere is used to detect if there is a collision between itself and the collider of any object that might be on the tile in question. If a collision occurs, and the node object that represents that tile within the grid array is set with an unwalkable Boolean flag, which is used later when it comes time to determine A\* paths.

Script 1: An excerpt from grid.cs. This calculates the number of tiles within the grid, tests for whether the tile is walkable or not and then ultimately utilizes the Gizmos API to draw a collection of wireframe boxes to demonstrate the walkable and non-walkable

private void Awake()

{

CreateGrid();

}

void CreateGrid() {

xStart = (int)bottomLeft.transform.position.x;

zStart = (int)bottomLeft.transform.position.z;

xEnd = (int)topRight.transform.position.x;

zEnd = (int)topRight.transform.position.z;

vCells = ((xEnd - xStart) / cellWidth);

hCells = ((zEnd - zStart) / cellHeight);

grid = new Node[hCells+1, vCells+1];

UpdateGrid();

}

public void UpdateGrid()

{

for (int i = 0; i <= hCells; i++)

{

for (int j = 0; j <= vCells; j++)

{

bool walkable = !(Physics.CheckSphere(new Vector3(xStart + i, 0, zStart + j), 0.4f, unwalkable));

grid[i, j] = new Node(i, j, 0, walkable);

}

}

}

void OnDrawGizmos()

{

if (grid != null)

{

foreach (Node node in grid)

{

Gizmos.color = (node.walkable) ? Color.white : Color.red;

Gizmos.DrawWireCube(new Vector3(xStart + node.posX, 0.5f, zStart + node.posZ), new Vector3(0.8f, 0.8f, 0.8f));

}

}

}

Once the position of the walls/unwalkable areas have been calculated, in order to ensure that calculations are done correctly, the script utilizes the Gizmos package from Unity to draw a collection of wireframe boxes on the maze which differentiate between walkable and unwalkable tiles. In order to achieve this, it determines the position of each tile by adding its index in the array, denoted by the posX and posZ variables, to the xStart and zStart variable respectively, this is done to determine the actual position of the tile within the scene as opposed to simply the relative location within the grid. From there, it determines if the tile is walkable or not by checking the walkable field of the node object associated with the field which had been previously set using the unwalkable mask. If the area is unwalkable, it sets the color of the cube that will be drawn to red, otherwise it chooses white to clearly differentiate between the two. After determining the color, it draws a cube with dimensions 0.8 x 0.8 x 0.8 in order to make it clearly visible on the tile and to allow for easier viewing of the individual tiles. These red and white wireframe boxes can be seen in Figure 3.

A picture containing indoor, sitting, small, light

Description automatically generated

Figure 3: Grid and shortest path generated by the A\* algorithm on a small maze. Red denotes unwalkable areas, white the walkable and green the shortest path between the two spheres.

As for calculating the actual path between two objects, the script first determines a list of neighbor tiles for a single space on the grid, these neighbors are the walkable tiles above, below and to the left and right of the current tile. Given this, the algorithm finds the quickest path from the start to the end. Beginning with the start node, the algorithm finds the total traveled distance to get to a neighbor by calculating the distance between the neighbor and the current node and adding it to the distance that had currently been traveled to get the current node. If the distance traveled to get to that neighbor from the current node is less than it had been on any other path, or if the neighbor had not yet been visited, the neighbor’s distance is recorded at the current calculated distance, the g cost, and has the current node marked as its parent node, used for later tracing the path. In addition, it also finds the h cost, the theoretical distance between that node and the goal, this is used in later decision making. Once all neighbors have been evaluated, it consults all visited nodes to find the one that has the, currently, smallest actual distance traveled and smallest theoretical distance and repeats the above process (Edpresso, 2020). This occurs until such time that it reaches the goal node, at which time the path is calculated by following the parent node of each node until it reaches the start node as seen in Script 2.

void FindThePath()

{

Node startNode = grid.NodeRequest(grid.start.transform.position);

Node goalNode = grid.NodeRequest(grid.goal.transform.position);

openList.Add(startNode);

while (openList.Count > 0)

{

Node currentNode = openList[0];

//cycle through all of the currently open nodes to find the one that has the smallest g and h cost

for (int i = 1; i < openList.Count; i++)

{

if (openList[i].fCost < currentNode.fCost || openList[i].fCost == currentNode.fCost && openList[i].hCost < currentNode.hCost)

{

currentNode = openList[i];

}

}

//The node, once selected, will never be ran through again

openList.Remove(currentNode);

closedList.Add(currentNode);

//if we've arrived at the end, find the actual path

if (currentNode == goalNode)

{

PathTracer(startNode, goalNode);

return;

}

//search for a continuation to this path amongst the current neighbors

foreach(Node neighborNode in grid.GetNeighborNodes(currentNode))

{

if(!neighborNode.walkable || closedList.Contains(neighborNode))

{

continue;

}

//calculate the total cost to get to that specific node from the start following the

//current path. If that distance is shorter, or the node has not been previously visited

//update the g and h cost of the node, along with the parent, ie, the current shortest path to

//get to the node

int calcMoveCost = currentNode.gCost + GetDistance(currentNode, neighborNode);

if(calcMoveCost < neighborNode.gCost || !openList.Contains(neighborNode))

{

neighborNode.gCost = calcMoveCost;

neighborNode.hCost = GetDistance(neighborNode, goalNode);

neighborNode.parentNode = currentNode;

if(!openList.Contains(neighborNode))

{

openList.Add(neighborNode);

}

}

}

*Keyboard Inputs*

Script 2: Excerpt from PathFinding.cs, this code executes the A\* algorithm, finding the shortest path between an object and its selected destination.

}

}

void PathTracer(Node startNode, Node goalNode)

{

List<Node> path = new List<Node>();

Node currentNode = goalNode;

while(currentNode != startNode)

{

path.Add(currentNode);

currentNode = currentNode.parentNode;

}

path.Reverse();

grid.path = path;

}

int GetDistance(Node a, Node b)

{

int distX = Mathf.Abs(a.posX - b.posX);

int distZ = Mathf.Abs(a.posZ - b.posZ);

return D \* (distX + distZ);

}

The player controls Pac-Man using either the W, A, S and D keys or the arrow keys to move him forward, right, backwards and left respectively. Every time the update method is called, Pac-Man calculates his next position based on where he wants to go, his current position and his speed which is multiplied by delta time in order to deal with frame rate issues. By pressing any of the directional button, this changes the direction of the destination in accordance with the button pressed. Until the player presses another directional button, Pac-Man will continuously move in that direction at a set speed until such time that he runs into a wall.

Regarding Pac-Man’s ability to detect a wall, this is done using a ray cast. Pac-Man uses a ray cast to detect objects in from of him, preventing him from repeatedly running into an object. First, in order to accomplish this, the script creates a ray object which is set to scan in the forward direction of Pacman, 0.25 units above the center of Pacman’s model. Using the Physics package, the script then checks one unit in front of Pacman if there is an object that contains the unwalkable layer mask, this information is then recorded in a RaycastHit object. If there is a collision with an object marked with the unwalkable layer mask, the script then consults the hit information to determine if what it hit has the “Wall” tag, which is used to denote walls and differentiate them from other unwalkable objects that are on the maze. If that object is marked as a wall, Pac-man’s attempt to move is invalidated, keeping him on the same tile, as seen in Script 3.

void Move()

{

transform.position = Vector3.MoveTowards(transform.position, destination, (speed \* Time.deltaTime));

if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow))

{

nextPos = Vector3.forward;

currentDirection = up;

}

else if (Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.DownArrow)

{

nextPos = Vector3.back;

currentDirection = down;

}

else if (Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.LeftArrow)

{

nextPos = Vector3.left;

currentDirection = left;

}

else if (Input.GetKeyDown(KeyCode.D) || Input.GetKeyDown(KeyCode.RightArrow)

{

nextPos = Vector3.right;

currentDirection = right;

}

if (Vector3.Distance(destination, transform.position) < 0.00001f)

{

transform.localEulerAngles = currentDirection;

if(Valid())

{

destination = transform.position + nextPos;

direction = nextPos;

}

}

}

bool Valid()

{

Ray myRay = new Ray(transform.position + new Vector3(0, 0.25f, 0), transform.forward);

RaycastHit myHit;

if(Physics.Raycast(myRay, out myHit, 1f, unwalkable))

{

if(myHit.collider.tag == "Wall")

{

return false;

}

}

return true;

}

Script 3: An Excerpt from Pacman.cs. This is the movement code for Pac-Man, adjusting the direction of motion and the rotation of the 3D model. This code executes until such time that Pac-Man's motion becomes invalid, causing him to remain on a single space.

*Ghost Movement and State*

In order to move around the maze, the ghosts utilize the A\* algorithm mentioned previously in order to locate the specific target for the ghosts. Like Pac-man, the ghosts utilize the Vector3.MoveTowards function to move toward a specific target at a set speed, which in the case of the ghosts is dependent upon its current state. As the ghost moves forward, it evaluates whether or not it has reached the current target space to move to by checking if it is 0.0001 units away from the next tile. Upon reaching the next tile, the path is recalculated using the A\*, and begins to move toward that next tile in the path. In addition, the ghost also properly orients itself to face the same direction as the next tile. This is done by subtracting the x and z distance of the current position from the next position, this is then fed into a collection of if else statements that determine as to what direction the ghost should face based on the sign of the x and z differences.

Ghosts have an enum based state (HOME, LEAVING\_HOME, CHASE, SCATTER, FRIGHTENED and GOT\_EATEN), this determines the behavior that it will exhibit. Every time the ghost object calls the update function, it checks the state that the ghost is currently in through the use of a switch statement at which point the movement of the ghost will be altered. When in the HOME state, ghosts move back and forward in the base located at the center of the maze. Ghosts contain a list of nodes that represent the end points of the path it will follow while in the base. If the ghost is not currently seeking out a target that isn’t within the list of nodes for the home base, it assigns the ghost’s target to be the first item in the list. After assigning it to move towards the new target, it loops through all of the other nodes, this is done to allow the ghost to change its target to the next item when it reaches the current target. When the loop reaches the index of the current target, it evaluates the distance between the ghost’s current position and that of the target though Vector3.Distance and checks if it is under 0.0001, same as when moving between tiles. If this condition is met, it sets the target to be the next node within the list , cycling back to the beginning if the node is the last in the list, and repeats the process until such time that the ghost is removed from the HOME state. These nodes are supplied to the ghost in the form of small, invisible cube objects placed on tiles within the map. These are then placed in the ghost’s public list of transformations, marking the exact locations that the ghost is meant to travel to. In addition to the looped movement, by setting the ghost into the HOME, state, the speed of the ghost is reduced to 1.5 as opposed to the normal 3, this was done for visual purposes. Both the SCATTER and FRIGHTENED states perform similar actions to that of the home state, with the main difference being the speed that these states set the ghost to as well as the target locations. In addition to modifying the behavior of the ghost, these states also adjust the material for the ghost, alternating between normal, frightened and eaten materials.

The CHASE state is simply where Pac-Man is set as the current target for the ghost, meaning that the A\* algorithm is working to create paths that lead to Pac-Man. At each tile, the ghost runs the A\* algorithm to update the path to coincide with the player’s current position. In addition, the CHASE state is where individual ghost’s personalities are put into account. Since all ghosts use the exact same script, their behavior is implemented through the use of another enum called ghosts (INKY, PINKY, BLINKY and CLYDE). If the ghost is given the CLYDE enum in Unity through the inspector, before making a movement, it evaluates the distance between its current vector position and that of Pac-Man. If the distance is less than 8, note this is not a distance of 8 tiles but is instead is a direct distance between Pac-Man and the ghost, the ghost uses the same method utilized by the SCATTER state to encircle the frightened nodes; however, it does not force the ghost into the SCATTER state. Once Pac-Man is further than 8 units away from the ghost, it will return to the original chase procedure. If a ghost is marked with the BLINKY enum, however, it does not have any special behaviors and simply chases Pac-Man whilst in this state. The other two ghosts utilize more complex behaviors.

The Pinky ghost is set to be 4 spaces ahead of Pac-Man, if possible. First, it creates a new transform object called ahead target, and then checks if the transform’s position is inside of the grid. This is done by subtracting the x and z components or the bottom left corner from the position of the transform’s x and z values. This then evaluates if the distance exceeds the total number of horizontal and vertical cells within the graph as well as to ensure that the distance is not below 0 and that the location in question is not a walkable location. If the transform is not found to be within the grid, the number of spaces it looks ahead is decremented, and the position of the transform is set to the current position of the Pac-Man target, plus the forward position times the look ahead, this is done to make the pinky ghost stay ahead of the Pac-Man without trying to find a location outside of the map. This continues until such time that the position of the transform lies within the bounds of the maze. After determining the new position, it sets the current target to the new target determined to be in front of Pac-Man, and then destroys the game object associated with the transform. At this point, the move function then moves the ghost to the new location in front of Pac-Man. This can be seen in Script 4.

Script 4: An exceprt from PathiFinding.cs. This determine's Pinky's behavior and searches for a target ahead of Pac-Man

void PinkyBehavior()

{

Transform aheadTarget = new GameObject().transform;

int lookAhead = 4;

//set the target

for (int i = lookAhead; i > 0; i--)

{

if(!grid.CheckInsideGrid(aheadTarget.position))

{

lookAhead--;

aheadTarget.position = pacManTarget.position + pacManTarget.transform.forward \* lookAhead;

} else

{

break;

}

}

aheadTarget.position = pacManTarget.position + pacManTarget.transform.forward \* lookAhead;

Debug.DrawLine(transform.position, aheadTarget.position);

currentTarget = aheadTarget;

Destroy(aheadTarget.gameObject);

}

Inky’s behavior is based on the idea of ambushing Pac-Man with the aid of the Blinky ghost. To begin, the InkyBehavior function creates three transform objects, blinkyToPacMan, a target and a goal. The blinkyToPacman transform is then instantiated as a new Vector3 which holds the distance between Pac-Man and the Blinky ghost, both are provided to the script via a public variable. The target, also instantiated as a Vector3, is instantiated with the position of Pac-Man plus the distance between Blinky and Pac-Man, this is so that Inky can attack Pac-Man from the opposite direction of Blinky. After determining the target location, Inky finds the closest non-wall location to the target. In order to do this, Inky calls the GetNearestNonWall function within the grid script. This function starts by creating 3 local variables, a min distance and an index for the minimum point in the 2D array that constitutes the grid. The function then iterates through each of the node objects in the grid, skipping over those marked unwalkable. For each walkable node, it then finds the distance between that node and the target determined by Inky utilizing Vector3.Distance. If the distance between those two points is less than the current minimum distance, which is initially set to 1000, the minimum distance and the indexes are updated. Once it has iterated through the entire grid, it will return a Vector3, denoting the position of the closest point to the target selected by Inky. This Vector3 is then set as the current target, and then the game objects associated with the transforms created by Inky are then destroyed, with the remainder of the motion being taken care of by the move function. The code for Inky’s behavior can be seen in Script 5.

void InkyBehavior()

{

Transform blinkyToPacman = new GameObject().transform;

Transform target = new GameObject().transform;

Transform goal = new GameObject().transform;

blinkyToPacman.position = new Vector3(pacManTarget.position.x - BlinkyLocation.position.x, 0, pacManTarget.position.z - BlinkyLocation.position.z);

target.position = new Vector3(pacManTarget.position.x + blinkyToPacman.position.x, 0, pacManTarget.position.z + blinkyToPacman.position.z);

goal.position = grid.GetNearestNonWall(target.position);

currentTarget = goal;

Debug.DrawLine(transform.position, currentTarget.position);

Destroy(target.gameObject);

Destroy(blinkyToPacman.gameObject);

Destroy(goal.gameObject);

}

Script 5: An excerpt from PathFinding.cs. This is used to control the behavior of the Inky ghost, setting it up to ambush Pac-Man with the help of Blinky.

In the GOT\_EATEN state, the ghost sets its target to be the first node in the list of home target list. Once the target has been set, the speed of the ghost is increased to the same as Pac-Man, allowing it to quickly return to the home base. Once the ghost is at a distance of 0.0001 from the target, it changes state back to the HOME state keeping it in the base until such time that the ghost is ready to be released.

State transitions are controlled through the use of a Timing function, seen in Script 5, which keeps track of the time that the ghost has been in a certain state, this is called every time the Update function for the object is called, immediately after the CheckState function described above. Each state is given two variables, a timer to count how long it has been in that state and a max time that it is supposed to be in that state. In the case of the home state, however, there is an additional Boolean variable called released, this denotes whether the ghost should be released from the base and if the timer should begin. If the timer is allowed, it adds the change in time between Update calls through Time.deltaTime. Once the compounded time has passed the designated threshold, the ghost’s state is change depending on the current state that it is in. If the ghost is in the HOME state, it is released into the chase state after 3 seconds, the CHASE state transitions to the SCATTER state after 20 seconds and the SCATTER state transitions back to the CHASE state after 7 seconds. The FRIGHTENED state also contains an additional check that the other states do not. When in the FRIGHTENED state, the ghost is vulnerable to Pac-Man and can be transitioned into the GOT\_EATEN state. As a result, it must check that it is not in that state before making the transition back to the CHASE state as the eaten ghost must return to the base before the game can continue.

Script 7: An excerpt from GameManager.cs. This is the timer used to transition ghosts between states. Each state has a specific timer, and once that time limit has been reached, the ghost will transition to its next state, resetting the timer as well.

void Timing()

{

UpdateStates();

if (chase)

{

currCTimer = currCTimer + Time.deltaTime;

if(currCTimer > cTimer)

{

currCTimer = 0;

chase = false;

scatter = true;

}

}

if (scatter)

{

currSTimer = currSTimer + Time.deltaTime;

if (currSTimer > sTimer)

{

currSTimer = 0;

chase = true;

scatter = false;

}

}

if (frightened)

{

if (currCTimer != 0 || currSTimer != 0)

{

scatter = false;

chase = false;

currCTimer = 0;

currSTimer = 0;

}

currFTimer = currFTimer + Time.deltaTime;

if (currFTimer > fTimer)

{

currFTimer = 0;

chase = true;

scatter = false;

frightened = false;

}

}

}

*Camera*

The game features an aerial, third person camera that is set to track Pac-Man as he moves across the maze. The camera is initially set to have a location of -1, 12, -8 with a 70° rotation about the x axis. This allows for the player to keep his/her eye on Pac-Man as well as see the surrounding map to be able to make later decisions. The camera is bound to Pac-Man, moving when he moves to allow the player to keep a view of Pac-Man whilst being forced to view the maze from so far away that it becomes impossible to have a clear view of what is going on within the maze. In the script, once the camera is created, it finds the distance between the x and z position of itself and of Pac-Man, this information is recorded for later use. When the Update function is called on the camera, it determines its new x, y and z positions based on the position of Pac-Man. The x and z positions are calculated by adding the distance calculated upon start to their respective components from Pac-Man’s current position, the y position is constant and does not change as a result. After calculating the new position of the camera, the camera is moved at a speed of 1.5 units using the Lerp function from Vector3 to gradually move the camera along with Pac-Man with a smooth transition. The speed for the camera, like that of both Pac-Man and the ghosts, is multiplied by deltaTime in order to account for varying framerates. When played, the camera slowly follows Pac-Man across the x and z axis of the maze, allowing the player to see what is immediately going on around Pac-Man without forcing them to stare at the entirety of the map at once. The entirety of this code can be seen in Script 8.

public Transform target;

float distX, distZ;

public float speed = 1f;

Vector3 pos;

void Start()

{

distX = transform.position.x - target.position.x;

distZ = transform.position.z - target.position.z;

}

void Update()

{

pos.x = target.position.x + distX;

pos.y = transform.position.y;

pos.z = target.position.z + distZ;

this.transform.position = Vector3.Lerp(transform.position, pos, speed \* Time.deltaTime);

}

Script 8: An excerpt taken from CameraTracking.cs. On start, the script records the distance between the camera and its target, Pac-Man in this instance. The camera then smoothly follows the target, keeping to the specified distance, through Lerp

*User Interface*

The User Interface is constructed through the use of a Canvas. This Canvas, designated UICanvas, contains a set of four names text boxes that display the score, level number, life count and the controls for the game. The first three text boxes are anchored to the upper left-hand corner of the screen and are large enough so that the player can see them with ease. Upon the player scoring points, earning a life or completing the level the text within the named text field will be updated to reflect the current value of the field that it represents. The remaining field, the controls, is anchored to the lower right corner and is kept intentionally small as to not make it the center of the player’s view; however, it is there in the event that the player needs to read it. This field is static and does not change according to the game. This UI can be seen in Figure 4.

A picture containing sitting, keyboard, computer

Description automatically generated

Figure 4: The main game complete with UI and mini-map

*Procedural Maze Generation*

Prior to the beginning of the level, the game builds the maze procedurally. In order to do this, the script used to generate the grid was given an additional function, instantiating instances of the wall prefab at specified locations. The function first determines the upper left corner of the maze, this is achieved by using the bottom left and top right corners that the grid utilizes to build itself. This corner is stored as a Vector3, utilizing the x position of the bottom left, the z position of the top right and the y position of the bottom left; however, as they are both at the same y location, it does not matter what corner’s value is used. After determining the corner, the function locates a file, maze.txt, which is comprised of a collection of2’s, 1’s and 0’s these are used to form the maze. The file is open and then read, line by line, into a string until such time that the end of the file is reached. Each line of the file is then iterated, looking at each individual character. If the character is a 2, the function will create a power pellet at that location, in addition, it will also spawn a grey tile to mark that specific section of the maze as a walkable space. If the character is a 1, the function will create an instance of the wall prefab provided to it via the instantiate function at the current location, which is initially the upper left corner previously stored within a Vector3. If a 0 is located in the current position, it has the possibility of spawning 2 items, a regular pellet and a tile. After reading the character, it increments the x position of the Vector3 by 1, the size of a single wall cube, and then continues. When the end of the line is reached, the z position of the Vector3 is decremented, and the x position is reset to that of the bottom left corner, so that the process can repeat. This function is called during the Awake function of the grid object, prior to the determination of the grid. This function can be seen in Script 9.

Script 9: An excerpt from grid.cs. This function reads in an input file line by line, character by character, and then builds a maze utilizing a wall prefab.

private void BuildWorld()

{

Vector3 currentMazePoint = new Vector3(this.bottomLeft.transform.position.x, this.bottomLeft.transform.position.y, this.topRight.transform.position.z);

string mazeLine;

StreamReader reader = new StreamReader("Assets/Scripts/maze.txt");

while((mazeLine = reader.ReadLine())!=null) {

for (int i = 0; i < mazeLine.Length; i++)

{

if (mazeLine[i] == '1')

{

Instantiate(wall, currentMazePoint, Quaternion.identity);

}

currentMazePoint = new Vector3(currentMazePoint.x + 1, currentMazePoint.y, currentMazePoint.z);

}

currentMazePoint = new Vector3(this.bottomLeft.transform.position.x, currentMazePoint.y, currentMazePoint.z - 1);

}

}

*Scoring*

The player is able to earn score using one of three methods, collecting a pellet, collecting a power pellet or eating an enemy ghost. The score is maintained by the GameManager object, which is also responsible for lives, level and win/loss conditions as well. Each pellet contains a sphere collider, which checks if Pac-Man has collided with it by checking if it has collided with an object that has the tag “Player.” If the pellet has successfully collided with Pac-Man, that specific instance of the pellet is the destroyed, calling the ReducePellet function from the GameManager. This function decrements the total number of pellets on the map, which is incremented by each pellet spawned, and increments the player’s current score by whatever point value the pellet contains as power pellets have a higher point value than regular pellets. This then triggers the update method in the UIManager, which replaces the string within the score, level and lives text boxes to display the current value of each of the variables. Once the player has collected all of the pellets, denoted by the pellet counter reaching 0, the level counter is incremented and the scene is reset using the ScreenManager class, which reloads the current scene so that the player can continue playing.

Unlike the pellets, the ghosts do not check for whether they have collided with Pac-Man or not, instead, Pac-Man checks if he has collided with a ghost using the same sphere collider method, checking if he had collided with an object marked with the “ghost” tag. Assuming the ghost is in the frightened state, Pac-Man calls the AddScore function of the GamaManager, which adds a set value to the game’s score. This then triggers the UIUpdate function of the UIManager, just as the pellets did.

In addition, to reward the player for playing well, if the player is able to earn the current level times 3000 points before completing the level and the scene resets, the player is given another life, which triggers the UIManager’s UIUpdate function. This is to keep the game interesting, while also making the player work harder on each new level.

*Combat*

The combat system for the game is primitive, but effective. When the game starts, Pac-Man is not able to attack the ghosts, losing a life if he makes contact with any of them. Pac-Man utilizes the OnTriggerEnter function to monitor his collisions with anything else that contains a collider. Upon collision, Pac-Man checks if the object that he has collided with has the ghost tag. If the object is a ghost, he then retrieves the PathFinding component of the ghost using the GetComponent function and stores it as a temporary value called pGhost. To determine what will occur after the collision, Pac-Man then checks the current state of the ghost, if the ghost is in neither the GOT\_EATEN or the FRIGHTENED state, the LoseLife function of the GameManager is called to decrement the value of the player’s current lives as well as store the current score and level in a static, external variable which is used for the game over screen, as well as calls the reset function for all 4 ghosts, by iterating through a list which is defined by the GameManager on start, and Pac-Man. These functions return the object in question back to its original position and direction which were selected at the beginning of the game. If the player’s life count reaches zero, the player is then shown the game over screen. The combat script can be seen below in Script 10.

void OnTriggerEnter(Collider other)

{

if(other.tag == "Ghost")

{

Debug.Log("I hit a ghost!");

PathFinding pGhost = other.GetComponent<PathFinding>();

if(pGhost.state == PathFinding.GhostStates.FRIGHTENED)

{

SoundEffects.instance.playGhostDied();

pGhost.state = PathFinding.GhostStates.GOT\_EATEN;

GameManager.instance.AddScore(500);

} else if (pGhost.state != PathFinding.GhostStates.GOT\_EATEN && pGhost.state != PathFinding.GhostStates.FRIGHTENED)

{

SoundEffects.instance.playPacDied();

GameManager.instance.LoseLife();

}

}

}

Script 10: An excerpt from pacman.cs. This is the trigger handler for Pac-Man. If he collides with a ghost, he checks the ghost's state and acts accordingly.

In order for Pac-Man to be able to eat the ghost, he must first collect a power pellet, which is a slightly larger version of the normal pellet. The pellet, like Pac-Man, utilizes the OnTriggerEnter function to check if it has encountered some sort of collision with another object that contains a sphere collider. If the object that it collided with contains the player tag, it performs its score mechanics and then sets the frightened value in the GameManager to true. On the next Update call in the GameManager, the ghost list, which is instantiated upon the GameManager starting by finding all items tagged with “Ghost” with the FindGameObjectsWithTag function, is iterated through, checking the state of each ghost and adjusting appropriately. When the frightened flag is set, assuming that the current ghost in the list is not in the GOT\_EATEN or in the HOME states, the state of the ghost will be set to FRIGHTENED. This then modifies the behavior of Pac-Man’s OnTriggerEnter function for the ghosts, changing the state of the ghost to GOT\_EATEN and then updating the score. Once the state of the ghost has changed, it will then act according to what is appropriate for that state.

*Start and Game Over*

The game features two additional scenes other than the main game scene, title main and game over. The main scene simply displays the original Pac-Man logo and a small message that asks the player to press start in order to play the game. The scene contains only a UI and a small script attached to the camera that utilizes the Input.OnKeyDown function to check if the player presses the space bar. Once the player has pressed the space bar, the function calls the SceneManager.LoadScene function, which then loads the main game scene. Te start screen can be seen in Figure 5.



Figure 5: The title screen which displays the original Pac-Man logo

The game over scene simply displays the words “Game Over” along with the player’s final score and the level that they had reached. The UI for this scene contains a small manager object that reads the score and level from the static fields of the ScoreHolder, which is updated after every player death, and displays them on the current screen by changing the text within the two text fields provided to the object. In addition, this contains another key listener which waits for the player to press the escape key, at which point it will call the Application.Quit function, quitting the game as a result. The end game screen can be seen in Figure 6.

A picture containing clock

Description automatically generated

Figure 6: The game over screen, this displays the text along with the player's final score and highest level reached.

*Sound*

The 3D version of Pac-Man contains some of the sounds found within the original game, such as the classic start and eating sounds. The start sound is played through the use of an audio source component that has been attached to the grid creation object. On awake, the object begins to play the audio source from a file, which is then picked up through the audio listener attached to the camera. The eating sound, which is played by Pac-Man, is also picked up via the camera as well. The eating sound, however, does not play at start like the intro sound. Upon the start of the game, Pac-Man retrieves the audio source component, stores it in a local variable and calls the Stop function to cease any audio that it may be playing. On each call to Pac-Man’s update function, if the canPlayAudio flag has not yet been set to true, it then utilizes a counter, similar to that which the ghosts use for changing states, to check if four seconds of time have passed, the length of the introductory music. When the timer has expired, the canPlayAudio flag is set to true, stopping the timer from progressing, and unlocking the audio player in the move function. On movement, if the conditions for Pac-Man to move are valid, an if statement checks if the canPlayAudio flag is set to true and the isPlayingAudio flag is set to false. If this condition is met, it calls the Play function of the audio source, and marks the isPlayingAudio flag as true. This prohibits the sound from being continually restarted on every movement, and as the source is set to loop by default, the audio will continue to play as Pac-Man moves. If the conditions are not met to allow Pac-Man to move, the isPlayingAudio flag is set to false, and the Stop function is called on the audio source, thus stopping the audio from playing while Pac-Man is stopped. Pac-Man’s audio code can be seen in Script 11. In addition to the start and eating sound, the game features background music that, while not from the original game, is still in 8-bit style. This is played through an empty game object that is located within the scene. Like Pac-Man, this also waits for the completion of the start up to begin playing; however, this contains a small Easter Egg, allowing the player to change the background music by pressing M (and reverting it back to normal by pressing N). Lastly, the game features two additional sound effects to denote if the player has either died or if the player has eaten a ghost. These are played by yet another empty game object, as to prevent them from interfering with background music and are started only through calls to the SoundEffects class’s static instance, as seen in Script 10.

Script 11: An excerpt from pacman.cs. This is the code used to control Pac-Man's audio source. The first code block is taken from the Move function, while the second is the counter used to determine if the intro sound has stopped playing.

if(Valid())

{

if (audioCanStart && !audioIsPlaying)

{

audio.Play();

audioIsPlaying = true;

}

destination = transform.position + nextPos;

direction = nextPos;

} else

{

audio.Stop();

audioIsPlaying = false;

}

}

void checkForAudioStart()

{

audioTimer += Time.deltaTime;

if (audioTimer >= playAudioTime)

{

audioCanStart = true;

}

}

*Cartoon Visuals*

The cartoon art style is provided by a shader which is used to modify the material of each 3D objects. This shader modifies the way that light interacts with the object, providing them with a semi glossy look to reflect a more cartoonish look. This shader was obtained from an online resource. Lighting is achieved through the use of a directional light accompanied by a light probe group. The probe group provides ambient lighting to the environment without having to bake the scene, which would not be possible due to its dynamic nature.

*Mini-Map*

In a modification from the original game’s formula, the 3D Pac-Man now includes a mini-map to help players make strategic decisions on the fly, as well as be able to see the all ghosts’ current positions. In order to achieve this, a second camera was brought into play that utilizes the orthogonal viewpoint as opposed to the perspective viewpoint utilized by the main camera, which is positioned 70 units directly above Pac-Man, and looks downward at a 90 degree x-axis rotation. From there, a raw image was added to the UI that is 256x256 in size and anchored to the lower right corner of the map, to which a blank texture is applied to, this will be used to display the image of the second game camera. The new camera is then given the texture as the output in the inspector, thus causing the image seen by the camera to be displayed within the raw image in the corner, thus creating a mini-map. In addition, a UI mask was added in order to give the min-map a circular appearance. A blank circle, sized to 256x256 just like the image, was added to the UI canvas and given a mask attribute. The raw image was then added to the circle as a child, thus displaying the image on the circle. Lastly, the mini-map was provided with a black circle outline in order to better differentiate it from the main game screen. In order to ensure that the mini-map remained focused on Pac-Man, the camera that records the image was given the CameraTracking.cs script in order to ensure that the camera follows Pac-Man whilst also ensuring that it remains at a fixed rotational point. Please consult Figure 4 and Script 8 for further information.

*Pause Menu*

In order to allow the player to pause the game, a small pause menu was added. In order to accomplish this, an empty game object was added to the scene with a small pause script. This script utilizes the update function to check if the player had pressed the space button. Upon pressing space, the function will then evaluate if the game is currently paused or not, denoted by a Boolean value. If the game is not currently paused, the function will set the paused flag to true and the timescale to 0, thus stopping all aspects of the game by stopping the game time, which all motion is based on. In addition, it sends a call to the UIManager to display the text PAUSED in a text box anchored to the center of the screen. If the player presses space again while the game is paused, it will reset the paused flag to false and then set the timescale back to 1, so that time can continue at the same rate as normal, thus resuming motion within the game. In addition, it makes another call to the UIManager to remove the text from the center of the screen. Please consult Figure 4 for further information.

*Animation*

To keep in line with the original game, Pac-Man has been given a moving mouth to show that he is eating as he moves. In order to do this, Pac-Man is given a child object that is nothing more than a small chunk taken from the original Pac-Man model which rotates around the center of Pac-Man. This rotation is achieved through a modification to the Pac-Man script which is responsible for his motion and interactions with ghosts. The rotation is handled through the moveMouth function, which utilizes the Rotation function from the mouth’s transform to rapidly rotate the mouth along the z axis. The mouth is given an initial rotation of -30 degrees, which signifies that the mouth is open. This rotation is then adjusted by the Rotate function which is given a Vector3 with an x and y value of 0 and a z value of mouthAngleOffsetAmount, which is a public variable used to denote how much the angle will be offset by. Initially this has a value of 10) and then adjusts the rotation of the mouth by that amount. This continues until the mouth reaches a z rotation of 30 degrees, which signifies that the mouth is now closed. Once the mouth has reached this point, as denoted by a float variable that is incremented by the offset reaching the threshold of 60, a Boolean flag is set flipped which multiplies the offset by -1, moving the mouth back towards -30 degrees and subtracting from the float variable. Once it returns to the open state, it reverses again. This continues for the duration of the game.

References

Edpresso. (2020). What is the A\* algorithm? Retrieved August 04, 2020, from https://www.educative.io/edpresso/what-is-the-a-star-algorithm

Pac-Man Fandom (2020). *Pac-Man Maze* [Digital Image]. Google Images.

https://www.google.com/search?q=pacman+maze&tbm=isch&ved=2ahUKEwix073TkNrqAhXHQa0KHVoyCU8Q2-cCegQIABAA&oq=pacman+maze&gs\_lcp=CgNpbWcQAzICCAAyAggAMgIIADICCAAyAggAMgIIADICCAAyAggAMgIIADICCAA6BwgjEOoCECc6BAgjECc6BAgAEEM6BQgAELEDOgcIABCxAxBDUIqqFlj94xZgnuUWaAFwAHgAgAFKiAGsBZIBAjExmAEAoAEBqgELZ3dzLXdpei1pbWewAQrAAQE&sclient=img&ei=s6kUX\_GtH8eDtQXa5KT4BA&bih=975&biw=1920&rlz=1C1RUCY\_enUS730US730#imgrc=d3nmZFybNMH4RM