Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

Construcción de Software y Toma de Decisiones

TC2005B.401

Ciclo 3. VideoJuegos

Profesor: Dr. Iván Olmos Pineda

Estudiantes:

Daniel Francisco Acosta Vázquez / A01736279

Diego García de los Salmones Ajuria / A01736106

Raúl Díaz Romero / A01735839

Rogelio Hernández Cortés / A01735819

Periodo Febrero - Junio 2023

7 / Mayo / 2023

Ciclo 3. VideoJuegos: Pacman en 3D

1. Propuesta

La propuesta de nuestro equipo fue realizar un Pacman en 3D con las siguientes cualidades:

- **Primera persona:** Decidimos que lo mejor sería desarrollar el pacman a partir de primera persona, esto para que sea lo suficientemente distinto del proyecto anterior, además de que se sentiría más inmersivo y tendría movimiento de cámara.
- **Inteligencia Artificial:** Los fantasmas tendrían algoritmos de inteligencia artificial para que persigan al jugador y representen un reto.
- **Sobrevivencia:** El objetivo de nuestro juego sería no capturar píldoras como en el pacman original, pero sobrevivir lo mejor posible, dando un puntaje que incrementa con cada segundo que sobreviva el jugador.
- **Fantasmas:** Los fantasmas se moverán por el laberinto utilizando una matriz de adyacencias y su inteligencia artificial decidirá a qué dirección ir para que se acerque lo más posible al jugador. Cuando un fantasma colisione con el jugador, el juego se cerrará y se mostrará por consola la puntuación obtenida.
- **Jugador:** El jugador se podrá mover libremente (en el eje x y z) dentro del laberinto. Habrá colisiones en el laberinto para evitar que el jugador salga del mismo. Podrá avanzar y retroceder al igual que realizar un movimiento Jaw con la cámara (de izquierda a derecha).

2. Control de movimiento y ambiente:

Para simplificación y tener un buen control del jugador, el jugador en si es la cámara, y se utilizan las siguientes funciones y variables para su movimiento y para el control del ambiente (dibujar y actualizar el ambiente):

- **float EYE_X, EYE_Y, EYE_Z,** estas variables representan las coordenadas x, y, z, de la cámara y por lo tanto, del jugador. Se inicializan en las coordenadas iniciales del jugador (8,0,0).
- **float WIDTH, HEIGHT,** variables que se utilizan para definir las dimensiones de la pantalla.
- **float FOVY, ZNEAR, ZFAR,** variables que se utilizan para el gluPerspective.
- **float Direction[3],** este arreglo es el vector de dirección de nuestro personaje/cámara.
- **float FEye[3],** este arreglo nos sirve para calcular la futura posición de la cámara/jugador, esto para rechazar el movimiento en caso de que ocasione una colisión.
- **void *objetos[NObjetos] (NObjetos = 42),** este arreglo es un arreglo de apuntadores a todos los objetos del juego, nos ayuda a realizar las tareas de chequeo de colisiones, update de los fantasmas y el draw de todos los objetos.
- **En la función init(),** inicializamos todos los objetos que sean de dos clases, clase wall() (los muros del laberinto), y clase ghost()/ghostR() (los fantasmas).

```
objetos[33] = new wall(-19, 0, 1, 7);
objetos[34] = new wall(-22, 0, 4, 1);

objetos[35] = new wall(-19, 25, 1, 2);
objetos[36] = new wall(-25, 14, 1, 9);
objetos[37] = new wall(-22, 12, 4, 1);
//fin paredes

//fantasmas
objetos[38] = new ghost(0, 1, 0.002, -28.0, -25.0, 1, 1.0, 0.0, 0.0, 0);
objetos[39] = new ghost(0, 1, 0.002, 28.0, 25.0, 1, 0.8, 0.3, 0.4, 0);
objetos[40] = new ghostR(0, 1, 0.002, -28.0, 25.0, 1, 0.3, 0.8, 1.0);
objetos[41] = new ghostR(0, 1, 0.002, 28.0, -25.0, 1, 0.9, 0.6, 0.0);
```

- **SpecialInput, Keyboard,** Estas funciones checan cuando el jugador presione las flechas direccionales o las teclas WASD, esto mediante el glutSpecialFunc(), y glutKeyboardFunc(). Al presionar las teclas W o la flecha de arriba, se verifica que la futura posición no ocasione colisiones y se actualiza la ubicación de la cámara/jugador. Lo mismo realizan al pulsar las teclas S o la flecha direccional hacia abajo.

```
switch(key){
                case GLUT_KEY_UP:
                    FEye_X += Direction[0];
                    FEye_Y += Direction[1];
                    FEye_Z += Direction[2];
                    FEye[0] = FEye_X;
                    FEye[1] = FEye_Y;
                    FEye[2] = FEye_Z;
                    if(!checkWallCollision(FEye)){
                    EYE_X += Direction[0];
                    EYE_Y += Direction[1];
                    EYE_Z += Direction[2];
                    CENTER_X = EYE_X + Direction[0];
                    CENTER_Y = EYE_Y + Direction[1];
                    CENTER_Z = EYE_Z + Direction[2];
                    }
                    break;
```

```
case 's':
case 'S':
    FEye_X -= Direction[0];
    FEye_Y -= Direction[1];
    FEye_Z -= Direction[2];
    FEye[0] = FEye_X;
    FEye[1] = FEye_Y;
    FEye[2] = FEye_Z;
    if(!checkWallCollision(FEye)){
        EYE_X -= Direction[0];
        EYE_Y -= Direction[1];
        EYE_Z -= Direction[2];
        CENTER_X = EYE_X + Direction[0];
        CENTER_Y = EYE_Y + Direction[1];
        CENTER_Z = EYE_Z + Direction[2];
    }
```

- **drawWalls(),** esta función recorre todo el array de objetos y crea un apuntador de tipo wall para dibujar cada pared del laberinto.

```
void drawWalls(){
    wall *aux;
    for(int i = 0; i < 38; i++){
        aux = (wall *)objetos[i];
        aux->draw();
    }
}
```

- **drawGhosts(),** esta función recorre los últimos elementos del array de objetos para dibujar a los fantasmas ya sea clase ghost() o ghostR(), creando un apuntador apropiado.

```
void drawGhosts(){
    ghost *aux;
    ghostR *aux2;
    for(int i = 38; i < NObjetos; i++){
        if(i == 40 || i == 41){
            aux2 = (ghostR *)objetos[i];
            aux2->draw();
        }
        else{
            aux = (ghost *)objetos[i];
            aux->draw();
        }
    }
}
```

- **updateGhosts(),** esta función recorre a los fantasmas en el arreglo de objetos y usando el apuntador apropiado, ejecuta el update(), de cada fantasma.
- **updateScore(),** esta función actualiza la variable score cada 1000 ejecuciones, de tal manera que el jugador obtiene 100 puntos cada segundo (aprox) que sobrevive en el juego. También se muestra la puntuación en el título del juego.

```
void updateScore(){
    time++;
    if(time>1000){
        score += 100;
        time = 0;

    }
    std::string t = "Pacman 3D   Score: "+std::to_string(score);
    glutSetWindowTitle(t.c_str());
}
```

- **display(),** esta función al ser la función idle del programa, determinado por glutIdleFunc(), ejecuta las instrucciones de dibujado de paredes, drawWalls(), dibujado de fantasmas, drawGhosts(), y el update de los ghosts, updateGhosts(), actualizar el puntaje, updateScore(), y el chequeo de colisiones entre fantasmas y el jugador/camara.

3. Clase ‘Walls’

Para el control y manejo de las colisiones del escenario, se creó la clase walls(). Esta clase es utilizada para inicializar todas las paredes del laberinto como objetos y hacer la detección de colisiones más sencilla.

Los atributos contenidos en esta clase son los siguientes:

- **float vertexCoords[24],** este arreglo nos ayuda definir las coordenadas de cada vértice de nuestro cubo que utilizaremos como pared.
- **float vertexColors[24]**, este arreglo nos ayuda a definir el color de cada vértice del polígono. Ya que el bloque será completamente azul, todos los valores son (0,0,1), azul en rgb.
- **int elementArray[24]**, este arreglo ayuda al dibujo del polígono dando el orden de los vértices del polígono de cómo se deben dibujar.
- **float Position[3]**, este arreglo es el arreglo de posición del bloque en coordenadas x, y, z. Ya que todos los muros estarán a la misma altura,pues no habrá variación vertical, los métodos que usan esta posición siempre manejan el valor [0] y [2], que son x y z respectivamente.
- **float Size[3]**, este arreglo contiene el factor de escalamiento que se utilizara para cada eje a la hora de dibujar la pared.
- **float radio**, se planeaba que esta variable fuera el radio de colisión de la pared, pero se encontró un método más eficaz por lo que la variable quedó sin utilizarse

Las funciones contenida en esta clase son las siguientes:

- **wall()**, es el constructor de esta clase.
- **void draw()**, esta función sirve para dibujar el objeto. Primero hace push a la matriz de modelado para posteriormente trasladar el objeto a las coordenadas dadas por el vector Position[3]. Después se realiza un escalamiento dado por el vector Size[3] para posteriormente dibujar el polígono y dar pop a la matriz de modelado.
- **Adicionalmente cuenta con getters y setters para obtener valores privados y establecer el valor de dichas variables.**

4. Clase ‘Ghost’

Para el control de los objetos fantasma se decidió generar una clase que contenga las diversas variables de control junto a las funciones necesarias para dibujar y actualizar la posición del objeto.

Los atributos contenidos en esta clase son los siguientes:

- **int direccion,** esta variable representa la dirección actual del objeto, este valor se calcula automáticamente y varía de valor dependiendo en qué tipo de intersección esté. En este caso manejamos 4 tipos de dirección para el objeto:
    - 1, dirección hacia arriba.
    - 2, dirección hacia abajo.
    - 3, dirección hacia derecha.
    - 4, dirección hacia izquierda.

- **int radioC, radioS**, esta variable hace referencia al radio que se utilizará para calcular colisiones en el caso de radioC y el radio del tamaño de la esfera que se va a dibujar por fantasma, radioS.
- **int iaLvl,** esta variable define que tan profundo deben buscar los fantasmas a la hora de realizar decisiones en base al algoritmo de inteligencia artificial.
- **float velocidad,** esta variable define la velocidad con la que irán los fantasmas.
- **float Position[3]**, esta variable almacena las coordenadas en el eje X, Y y Z del fantasma.
- **float color[3]**, este arreglo mantiene el color, en rgb, del fantasma.
- **GLUquadric *quad,** un apuntador utilizado para poder dibujar a los fantasmas como esferas.

Las funciones contenida en esta clase son las siguientes:

- **ghost(int, int, float, float, float, float, float, float, float, float, int)**, esta función sirve como el constructor del objeto ghost.
- **void draw()**, esta función sirve para dibujar el objeto fantasma, para ello hace push a la matriz de modelado, establece el color usando el arreglo color[3], y se dibuja la esfera en las coordenadas dadas por el arreglo Position[3]. Finalmente se hace pop a la matriz de modelado.
- **void update(int, int)**, esta función sirve para actualizar tanto la posición del fantasma como la dirección de su movimiento. Esto lo hace pidiendo el tipo de celda al igual que la dirección. primero verifica que la dirección no sea 0. En caso de que sí, establece una dirección inicial random. En caso contrario, la dirección se actualiza a la dirección pasada como parámetro (que se decide en el algoritmo de ia). Finalmente, se actualiza la posición en base a la dirección a la que se está yendo.

También está la clase 'ghostR()' que solo cambia en dos cuestiones. La clase ghostR no tiene el parámetro iaLvL, y su update solo requiere la celda, pues dependiendo de la celda, calcula su siguiente movimiento de forma aleatoria, contrario a la clase ghost que utiliza un algoritmo de ia.

5. Detección de colisiones

Para realizar la detección de colisiones, se utilizan dos funciones, una que detecta las colisiones entre jugador y paredes, y la otra que detecta colisiones entre fantasmas y el jugador.

- **checkWallColision(float p[3]),** esta función utiliza un sistema similar a la función drawWalls(), ya que recorre todo el array de objetos y por cada uno de ellos verifica dos cosas. La primera es que el jugador/cámara no pueda salir del laberinto, esto verificando que la posición futura (con FEye) no está fuera de los límites del laberinto. La segunda, verifica que la posición futura (igual con FEye) no esté dentro de un bloque. Para esto verifica que la posición futura no esté entre el rango de la

posición x de la pared +- su tamaño en x (dado por Size[0]) y que no esté en el rango de la posición z +- el tamaño en z (dado por Size[2]).

- **checkPlayerCollision(),** esta función lo que hace es checar los últimos objetos del array de objetos, los fantasmas, y calcula la heurística en su posición. Esto lo hace para verificar la distancia entre cada fantasma y el jugador. Si la distancia es menor o igual que el radio de colisión, el juego termina y se muestra tu puntuación final por consola. Ya que esto se ejecuta en la idle function display(), cada ciclo se ejecuta por lo que siempre se checa si un fantasma ha tocado al jugador

6. Inteligencia Artificial

Para implementar inteligencia artificial se utilizó el algoritmo minimax. Se escogió este algoritmo ya que era el más fácil de implementar en un juego de naturaleza como Pacman. Esto debido a que otros algoritmos como BFS o A* requieren una meta final, la cual es difícil de proporcionar dado que el pacman constantemente se mueve y no en coordenadas exactas. Sin embargo, en un algoritmo Minimax, podemos evaluar una heurística (como la distancia entre el fantasma y el jugador) para evaluar el valor de cada movimiento que pueda realizar el fantasma.

Para la implementación de este algoritmo,se utilizaron 3 diferentes funciones.

- **int minimax(float x, float z, int c, int dir, int h),** Esta función es la que se manda a llamar para el algoritmo. Se pasan las coordenadas x y z del fantasma en el punto que se mande a llamar al igual que la celda en la que se encuentra (int c) la dirección que lleva (ya que al no poder regresar, las posibilidades de cada intersección son distintas dependiendo de la dirección por donde se llegue a dicha intersección) y la variable h, que define que tantos hijos (o movimientos futuros) se deben tomar en cuenta y visitar. Si h es 0, se evalúa la heurística en todos los posibles puntos/intersecciones a las que se puede llegar y se consigue la menor y se regresa la dirección correspondiente. Si h no es 0, entonces se manda a llamar a la función maxIA, con h-1, de cada uno de estos siguientes puntos.
- **float maxIA(float x, float z, int c, int dir, int h),** si h es igual a 0, entonces se evalúa la heurística de los posibles puntos a los que se puede llegar de esta intersección y se regresa el mayor. Si h no es 0, se manda a llamar a la función mínIA en cada uno de estos posibles puntos.
- **float minIA(float x, float z, int c, int dir, int h),** muy similar a la función minimax, solamente que no regresa la dirección pero la heurística evaluada en los posibles puntos en caso de que h sea 0, si no se manda a llamar a la función maxIa en dichos puntos.

Adicionalmente, para poder obtener los futuras intersecciones a las que se puede llegar desde un punto se utilizaron las funciones moveUp(), moveDown(), moveLeft(), moveRight(), las cuales movían las coordenadas X o Z hasta la siguiente intersección válida.

7. Control de cámara

Para el manejo de cámara, se utilizan unas cuantas funciones

- **float RadToDeg, DegToRad,** estas funciones convierten de radianes a grados y viceversa. Utilizado para modificar el vector de dirección dependiendo hacia que dirección estamos observando.
- **void LookAt(),** esta función lo que hace es actualizar el vector de dirección en base a la variable theta para cuando se haga la suma o resta del vector de dirección con el de posición, se actualice la posición correctamente dependiendo de hacia dónde se está moviendo la cámara. También actualiza el center de la cámara para que voltee hacia donde le estamos indicando y siempre vea "hacia delante"

Link al video: https://youtu.be/o06wxC3e0pk

8. Reflexiones

- **Daniel Francisco Acosta Vázquez (A01736279)**: Este proyecto me dejó mentalmente y físicamente exhausto, pero fue increíblemente entretenido. En sí, muchos temas vistos en clase no los llegué a comprender en un 100% hasta que se realizó este proyecto. Desde el manejo de la cámara hasta el dibujado de formas tan simples como lo son esferas o bloques. Trabajar en un ambiente 3D es sin dudas, más complicado en un inicio, pues todo requiere de mayor preparación, o incluso cosas como la inicialización de todo el laberinto toma mucho tiempo. Sin embargo, una vez empiezas a avanzar en el proyecto, pequeños cambios o implementaciones menores se realizan mucho más rápido que cuando trabajamos en un ambiente 3D.

Sin dudas, este proyecto me deja un gran entendimiento de como herramientas de alto nivel trabajan diferentes aspectos de un ambiente 3D, y me hace valorar que estas herramientas facilitan muchas cosas, como por ejemplo el agregar interfaces, o agregar modelos 3D de maneras sencillas, para no recurrir a figuras básicas

De los mayores retos presentados fue sin duda alguna la implementación de IA, por lo que es otro aspecto que agradezco se simplifique con muchas otras herramientas. Implementarlo no fue fácil, pues se probaron muchas otras opciones que al final no funcionaban como esperábamos. Para implementar una buena IA no es solo necesario saber el juego a la perfección, si no saber también cómo funciona a la perfección y cuales son los límites del sistema.

Finalmente, puedo decir que el desarrollo de videojuegos es un proceso demasiado más complejo de lo que uno piensa en un inicio. Es un proceso arduo de prueba y error y de programación intensiva. Sin embargo , me gustó mucho la experiencia. Es un proceso lleno de obstáculos que debes superar como desarrollador y que te recompensa cada vez que ejecutas el programa y ves tu avance reflejado en la ventana emergente. Es un proceso

complicado, pero divertido, como un videojuego mismo. Sin dudas, este es un camino el cual deseo seguir explorando.

- **Diego García de los Salmones Ajuria (A01736106)**: En este proyecto llevamos nuestras habilidades e ideas a un nuevo nivel con el desarrollo de un videojuego en 3D, en este caso, trabajamos nuevamente la idea del videojuego Pac-Man pero ahora en 3D, una gran ventaja fue que ya conocíamos la esencia de este videojuego por haberlo trabajado en 2D en el proyecto anterior, lo cual contribuyó a que lográramos desarrollar un buen Pac-Man 3D. Todo este módulo de desarrollo de videojuegos me permitió conocer todo lo que conlleva la programación de un videojuego, lo que me hace apreciar lo muy complejos que son los procesos que se llevan a cabo para crear los videojuegos que conocemos y hemos jugado alguna vez. Sin duda alguna, fue una materia de muchos aprendizajes para mí.

- **Raúl Díaz Romero (A01735839)**: Este proyecto, similarmente al proyecto en 2D, fue sumamente interesante y revelador en todos sus aspectos, pues comprender todo el funcionamiento que existe detrás de un videojuego 3D fue impresionante e impactante.

  Mediante este proyecto, pude comprender la cantidad de dedicación y tiempo necesarios para desarrollar un videojuego 3D, y cómo es aún más desafiante que crear un videojuego 2D. Para crear un videojuego atractivo y funcional se requieren habilidades técnicas avanzadas y una gran dosis de creatividad. En el caso de nuestro proyecto, tomamos como ejemplo el icónico videojuego Pac-Man y le dimos un giro interesante al transformarlo en un formato 3D.

  Durante este proyecto, adquirí varios aprendizajes importantes, tales como la habilidad de crear y manipular objetos en 3D, rotar y trasladar dichos objetos, generar colisiones con objetos en 3D, implementar algoritmos de inteligencia artificial y manipular la cámara del jugador en diversos aspectos, todo utilizando OpenGL y C++.

  La integración de la inteligencia artificial en nuestro proyecto resultó ser un desafío significativo, especialmente en la programación de los movimientos de los fantasmas. Fue necesario considerar factores como la estructura del tablero, las posibles rutas disponibles, la ubicación actual del fantasma y del Pac-Man. A pesar de esto, se logró con éxito implementar la IA en nuestro proyecto.

  Para mejorar el proyecto, considero que sería beneficioso añadir más detalles, como la implementación de algoritmos de búsqueda más complejos y eficientes para los fantasmas a medida que se avance en los niveles, mantener un historial del puntaje más alto de los jugadores, incluir efectos de sonido en el videojuego y generar múltiples niveles para agregar más variedad y desafío.

  En conclusión, este proyecto fue desafiante, pero al mismo tiempo gratificante, ya que ver el resultado final completamente funcional y dentro de los parámetros del reto, fue

muy satisfactorio. Adquirir nuevos conocimientos a lo largo del proceso también resultó en un sentimiento bastante agradable.

- **Rogelio Hernández Cortés (A01735819)**: Está segunda entrega del módulo representó un reto mayor, a pesar de que nuestra base fue el mismo juego que hicimos en la entrega 2D, la implementación de los elementos para que fuera 3D fueron bastante interesantes y nada triviales.

  Este proyecto nos demostró de manera práctica que el desarrollo de videojuegos dentro de las ramas de computación a este nivel de programación necesita un amplio conocimiento en lógica computacional, involucrando así; amplios conocimientos en estructura de datos, programación orientada a objetos, etc; a su vez, también necesita un buen manejo sobre conceptos elementales de matemáticas.

  A partir de esto, es que así como el 2D, también el desarrollo de este proyecto nos ayudó a reforzar los conocimientos que ya teníamos sobre programación y además nos permitió tener un primer acercamiento a lo que es el desarrollo con IA.

  En conclusión, a pesar de que aún no se ha encontrado cómo involucrar este módulo en el bloque, espero si se llegue a encontrar en un futuro la unión, me parece muy importante tenerlo en este semestre, ya que, de lo contrario no habría manera de reforzar lo que se aprende en 3er semestre y por otro lado, fue muy interesante este acercamiento que tuvimos al desarrollo de videojuegos, nos permite tener un mejor panorama de hacia dónde nos queremos dirigir al terminar la carrera.