Benemérita Universidad Autónoma de Puebla.

EXAMEN DE PRIMER PARCIAL.

Graficación.

NOMBRE: Pedro Vargas Arenas

Matrícula: 201734553

INTRODUCCIÓN.

A manera de concluir el primer parcial y como prueba de poner en practica todo lo aprendido en este lapso de tiempo, se tiene el proyecto final de parcial. Este proyecto consiste en realizar un escenario que muestre objetos compuestos por figuras elementales (círculos, cuadrados, triángulos) y en estos objetos se aplican las operaciones matriciales para obtener su respectiva matriz de modelado. Las matrices manejadas son de: Rotación, traslación, escalamiento. Estas matrices se multiplican en un orden determinado para conseguir lo antes dicho.

También se incluyen las técnicas de graficado nativo que se han estado manejando, el uso de clases y objetos para una mejor manipulación del código y el uso general de las figuras unitarias sobre los cuales se manejan con su correspondiente matriz de modelado.

OBJETIVOS.

Aplicar los conocimientos adquiridos sobre clases y objetos en C++.

Implementar las matrices de operaciones (rotación, traslación, escalamiento).

Uso de objetos unitarios.

Practicar las técnicas de graficado vistas en clase.

Poner en práctica el concepto de matriz de modelado.

Reutilizar códigos previamente realizados como actividad.

Aplicar operaciones con matrices y vectores.

Construcción de elementos compuestos por figuras simples.

MARCO TEÓRICO.

¿Qué es una matriz de modelado?

Una matriz de modelado es una estructura que contiene las transformaciones necesarias para un vector de puntos. La matriz de modelado usualmente contiene ya operadas matrices de traslación, rotación y escalamiento, al estar contenidas en una sola matriz, hace más sencilla la operación de estas con los puntos a transformar.

Clases y objetos en C++.

En la programación orientada a objetos, los datos y el código que actúan sobre los datos se convierten en una única entidad denominada clase. La clase es una evolución del concepto de estructura, ya que contiene la declaración de los datos. Pero se le añade la declaración de las funciones que manipulan dichos datos, denominadas funciones miembro o, también, métodos. Además, en la clase se establecen unos permisos de acceso a sus miembros. Por defecto, en una clase los datos y las funciones se declaran como privados

reservando espacio en memoria, luego lo inicializamos, guardando en memoria un dato o un conjunto de datos. Posteriormente usamos esos datos guardados en memoria. Pero un objeto de una determinada clase no solamente sirve para guardar datos, sino que además puede manipular dichos datos, a través de las llamadas a las funciones miembro.

Matrices de operación (rotación, traslación, escalamiento).

Escalamiento.

El escalamiento permite cambiar el tamaño de un objeto expandiéndolo o contrayéndolo en sus dimensiones.

En 2D implica el cambio de tamaño de un polígono, donde cada punto $p = (x_1, x_2)$ es transformado por la multiplicación de dos factores de escalamiento s_1 y s_2 a lo largo de sus ejes respectivos. Por lo que matriz se define como:

$$\begin{matrix} Ex & 0 & 0 \\ 0 & Ey & 0 \\ 0 & 0 & 1 \end{matrix}$$

Traslación.

La traslación permite desplazar un objeto a lo largo de sus dimensiones, como resultado se obtiene un cambio de posición.

En 2D implica el desplazamiento de un polígono, donde cada punto $p = (x_1, x_2)$ es trasladado d_1 unidades en el eje X_1 y d_2 unidades en el eje X_2, de esta forma, las coordenadas del nuevo punto se obtienen con la siguiente matriz:

$$\begin{matrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{matrix}$$

Rotación.

La rotación permite girar un objeto sobre un eje de rotación, dado un valor de ángulo de rotación θ y su dirección.

En 2D se lleva a cabo alrededor de un punto, se especifica el ángulo de rotación θ, y el punto de rotación (pivote) sobre el cuál el objeto será rotado. Utilizando coordenadas polares, el punto $p = (x_1, x_2)$ se puede escribir como $p = (r,\ \theta)$ y el punto $p' = (x_1', x_2')$ como $p' = (r,\ \phi + \theta)$. Su representación matricial es:

$$\begin{matrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{matrix}$$

DESARROLLO Y CONOCIMIENTOS ADQUIRIDOS.

Para desempeñar esta práctica, se requirieron códigos y algoritmos previamente realizados como actividad, entre estos está: Algoritmo de Bresenham, las clases Punto y Línea para su graficado, algoritmo para representar círculos, entre otros.

Como se pide la representación de círculos, cuadrados y triángulos, el algoritmo para esto se tiene dentro de las clases:

- Círculo.
- Cuadrado.
- Triángulo.

Dentro de cada clase, se tienen métodos para asignar los puntos de la figura a un arreglo de tipo Punto y así acceder a estos de una manera más simple. Los puntos que se asignan son los unitarios.

- En clase *Círculo.*

```
void Circulo::unitarioCirculo(){

    float i;
    int n = 0;

    for(i = 0; i < 2*(atan(1)*4); i += 0.128){
        puntos[n].setValues(1*cos(i), 1*sin(i));
        n++;
    }
}
```

- En clase *Cuadrado.*

```
void Cuadrado::unitarioCuadrado(){

    puntosC[0].setValues(0, 0);
    puntosC[1].setValues(1, 0);
    puntosC[2].setValues(1, 1);
    puntosC[3].setValues(0, 1);
}
```

- En clase *Triángulo.*

```
void Triangulo::unitarioTriangulo(){

    puntosT[0].setValues(0, 0);
    puntosT[1].setValues(1, 0);
    puntosT[2].setValues(0.5, 1);
}
```

De igual manera, se tienen otras funciones que crean rectángulo (en clase *Cuadrado*), óvalo (en clase *Círculo*) y un triángulo rectángulo (en clase *Triángulo*).

- En clase *Círculo.*

```
void Circulo::unitarioCirculo1(){

    float i;
    int n = 0;

    for(i = 0; i < 2*(atan(1)*4); i += 0.128){
        puntos[n].setValues(1*cos(i), 0.5*sin(i));
        n++;
    }
}
```

- En clase *Cuadrado.*

```
void Cuadrado::unitarioCuadrado1(float m){

    puntosC[0].setValues(0, 0);
    puntosC[1].setValues(1, 0);
    puntosC[2].setValues(1, m);
    puntosC[3].setValues(0, m);
}
```

- En clase *Triángulo.*

```
void Triangulo::unitarioTriangulo1(){

    puntosT[0].setValues(0, 0);
    puntosT[1].setValues(1, 0);
    puntosT[2].setValues(0, 1);
}
```

Al igual se cuenta con funciones que asignan la matriz de rotación, traslación y escalamiento respectivas. Para asignar estas matrices a cada figura, se requiere una clase Ma*triz* y esta se instancia dentro de las clases antes mencionadas para manejar las matrices (la instancia se llama *matrizOperacion*).

Esto se realiza en todas las clases.

```
void Triangulo::setMatrizEscalamiento(float e){
    matrizOperacion.setEscalacion(e);
}

void Triangulo::setMatrizRotacion(float grados){
    matrizOperacion.setRotacion(grados);
}

void Triangulo::setMatrizTraslacion(float x, float y){
    matrizOperacion.setTraslacion(x, y);
}
```

Esto se hace para que, al multiplicar la matriz de modelado por los puntos unitarios, se tenga exactamente los puntos ya transformados para graficar.

Como solamente se quieren el *X* y *Y* de los puntos, estos se guardan en un arreglo de flotante junto con un 1.

```
void Triangulo::puntosTriangulo(int i){

    puntosF[0] = puntosT[i].getX();
    puntosF[1] = puntosT[i].getY();
    puntosF[2] = 1;
}
```

Los puntos resultantes se regresan y guardan dentro del mismo arreglo de tipo Punto. Todo este proceso se repite en cada clase, es decir, para cada figura.

```
void Triangulo::pTriangulo(int i){

    puntosT[i].setValues(matrizOperacion.puntosR[0], matrizOperacion.puntosR[1]);
}
```

Dentro de la clase *Matriz* se definen las matrices para rotar, trasladar y escalar dependiendo de la rotación, traslación y escalamiento que se quiera, esto se recibe como parámetro al inicializarlas.

Para la matriz de rotación:

```
void Matriz::setRotacion(float r){

    rt[0][0] = cos(r);
    rt[0][1] = -sin(r);
    rt[0][2] = 0;
    rt[1][0] = sin(r);
    rt[1][1] = cos(r);
    rt[1][2] = 0;
    rt[2][0] = 0;
    rt[2][1] = 0;
    rt[2][2] = 1;
}
```

$$\begin{matrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{matrix}$$

Para la matriz de traslación:

```
void Matriz::setTraslacion(float x, float y){

    tl[0][0] = 1;
    tl[0][1] = 0;
    tl[0][2] = x;
    tl[1][0] = 0;
    tl[1][1] = 1;
    tl[1][2] = y;
    tl[2][0] = 0;
    tl[2][1] = 0;
    tl[2][2] = 1;
}
```

$$\begin{matrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{matrix}$$

Para la matriz de escalamiento:

```
void Matriz::setEscalamiento(float e){

    ec[0][0] = e;
    ec[0][1] = 0;
    ec[0][2] = 0;
    ec[1][0] = 0;
    ec[1][1] = e;
    ec[1][2] = 0;
    ec[2][0] = 0;
    ec[2][1] = 0;
    ec[2][2] = 1;
}
```

$$\begin{matrix} Ex & 0 & 0 \\ 0 & Ey & 0 \\ 0 & 0 & 1 \end{matrix}$$

Se cuenta con una función que las multiplica en el siguiente orden: (Traslación x Rotación) x Escalamiento. Esta función recibe el vector de puntos unitarios, una vez que se efectúan las operaciones, la matriz resultante se multiplica por los puntos y así se obtiene los puntos para finalmente dibujarlos.

```
void Matriz::operar(float* p){
```

Estos puntos se envían a la función que traza una línea. La función es propia de la clase de la figura a ilustrar.

```
for(i = 0; i < 2; i++)
    dibujarTriangulo(getPunto(i).getX(), getPunto(i).getY(), getPunto(i+1).getX(), getPunto(i+1).getY());
dibujarTriangulo(getPunto(2).getX(), getPunto(2).getY(), getPunto(0).getX(), getPunto(0).getY());
```

Todo este proceso se realiza en una función que tienen todas las clases de figuras básicas llamada *mostrar*.

```
void Triangulo::mostrar(float escala, float rota, float x, float y){
```

Para iniciar con la construcción del escenario, se crearon más clases, las cuales contienen una serie de figuras básicas que están colocadas de tal manera que asemeja la figura que se quiere.

Para lograr esto, se manipulan las matrices de los cuadrados, círculos y triángulos a necesitar, es decir, el escalamiento, los grados de rotación y la posición en donde se quiere la figura se mandan como parámetro a la función encargada de generarla.

Las clases que se encargan de esto son:

- Escenario.
- Pirámide.
- Nave.
- Luna.
- Alienígena.

Las clases *Nave*, *Pirámide*, *Luna* y *Alienígena* se encargan de hacer la figura que llevan por nombre. Para suscitar la nave, se necesitaron operar objetos de tipo *Rectángulo*, *Cuadrado* y *Triángulo*, es decir, modificar la escala, la rotación y la posición donde va a estar.

```
class Nave{

    private:
        Linea linea;
        Punto punto;
        Cuadrado cuadrado;
        Triangulo triangulo;
```

Para la pirámide, sólo se requirió un objeto de tipo *Triángulo* y una *Línea*.

```
class Piramide{

    private:
        Linea linea;
        Triangulo triangulo;
```

Para la luna, se necesitó escalar un objeto de tipo *Círculo*.

```
class Luna{

    private:
        Circulo circulo;
```

Finalmente, para el alienígena, se necesitaron objetos de *Línea*, un objeto de tipo *Circulo* para crear un óvalo y un triángulo.

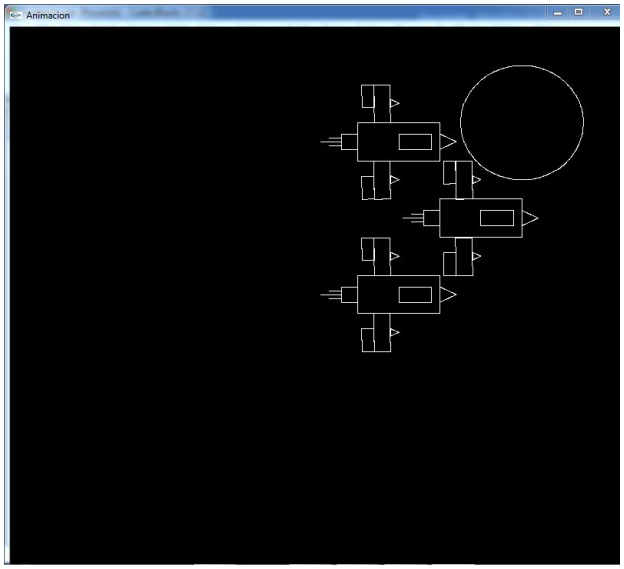
```
class Alienigena{

    private:
        Linea linea;
        Circulo circulo;
        Triangulo triangulo;
```

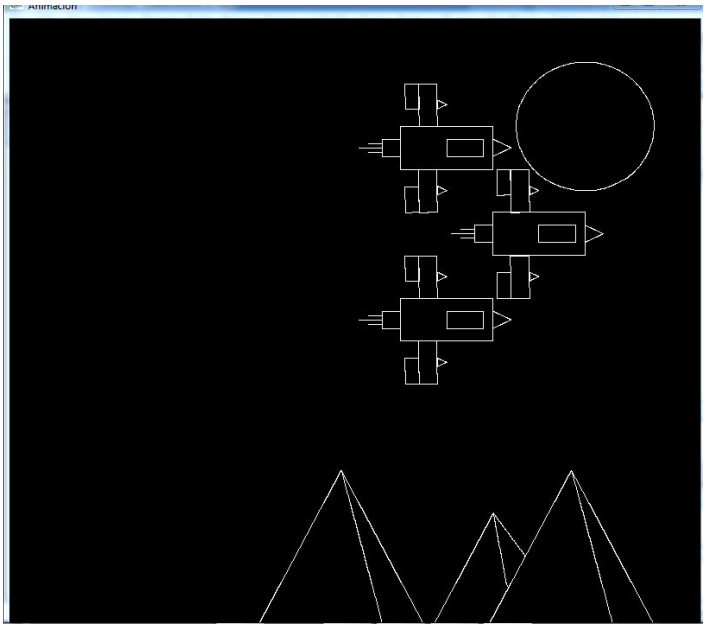
Una vez que las estructuras están creadas, estas funciones se llaman dentro de un ciclo en una función de la clase *Escenario*. Esta función de la clase *Escenario* se llama en la función *display* que está en la clase *main*. Gracias a una función propia de *OpenGL*, se manda a pantalla todo lo anterior.

PRUEBAS.

Debido que se iban a plasmar muchas figuras a la vez, todo se fue haciendo por pasos. Primero se realizó la estructura de la nave y esta misma se mandó a dibujar dos veces solamente modificando la posición que tendría, ya que si no se realizaba esto, la misma figura iba a estar sobrepuesta.

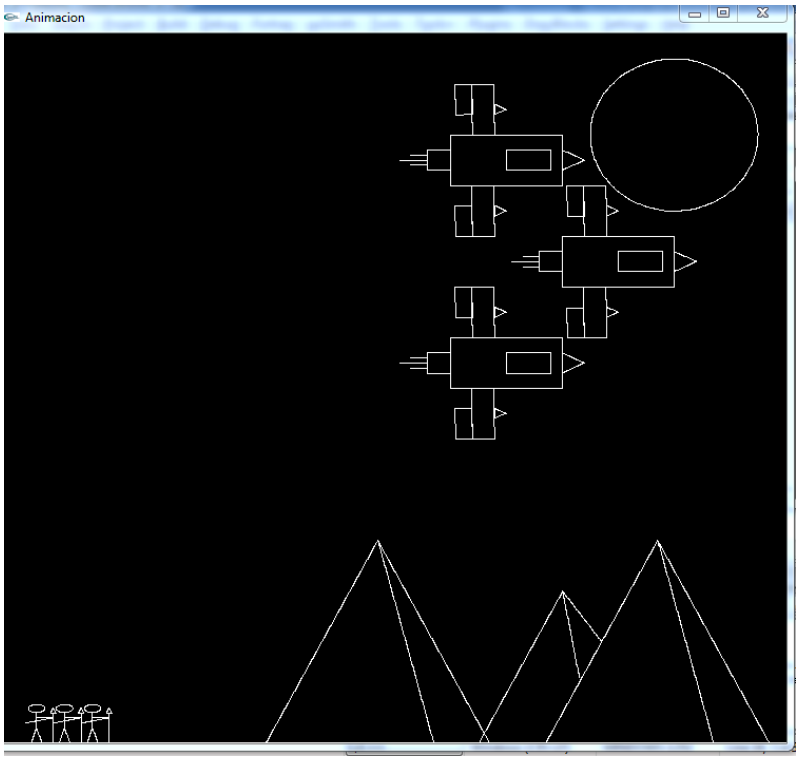


Una vez concluido el paso anterior, se procedió con el dibujado de las pirámides. De igual manera que con las naves, se repitió la estructura, pero con diferente traslación, la única estructura diferente es la segunda pirámide, ya que se realizó con líneas.



Lo único que restaba era colocar los alienígenas en el mismo eje *Y* de las pirámides.

Resultado Final:



CONCLUSIONES.

Como el proyecto exigía cierta dificultad, hubo muchas complicaciones, entre estas, hubo con el manejo de los objetos unitarios, la rotación de los objetos, pero en particular, con el manejo y operación de las matrices ya que tenía que realizarse de una sola manera y, además, multiplicarse con los puntos, esto se tenía que hacer con u buen manejo de índices y ciclos, porque es la forma más óptima de operar estas estructuras y gracias a que todo el proyecto está basado en clases, esto tuvo una solución eficaz y sin hacer operaciones más complejas como el manejo de doble apuntador.

Con los conceptos que analizaron durante las clases se facilitó en cierta manera el uso de las clases, cómo manipular las líneas y los puntos que se necesitaban para dibujar, la asignación de los puntos, entre otros. Lo que requirió más esfuerzo fue cómo crear y manipular la matriz de modelado de cada figura porque en un inicio no se veía claramente cómo se generaba y la manera en que se aplicaba con los puntos, pero con las ideas que se dieron en las clases previas a la realización de este proyecto, se llegó a una solución.

Gracias a la elaboración de este programa se pudo un poco de la complejidad que tiene este lenguaje, además de que se logró una mejor compresión los conceptos en cuanto al dibujado en 2D.

BIBLIOGRAFÍA.

https://elbauldelprogramador.com/clases-y-objetos-introduccion/

http://www.opengl-tutorial.org/es/beginners-tutorials/tutorial-3-matrices/

http://catarina.udlap.mx/u_dl_a/tales/documentos/mcc/cruz_m_ia/capitulo3.pdf}

Mathematics for Computer Graphics, Second Edition, John Vince. Springer.

Introduction to Computer Graphics. Frank Klawonn. Springer.