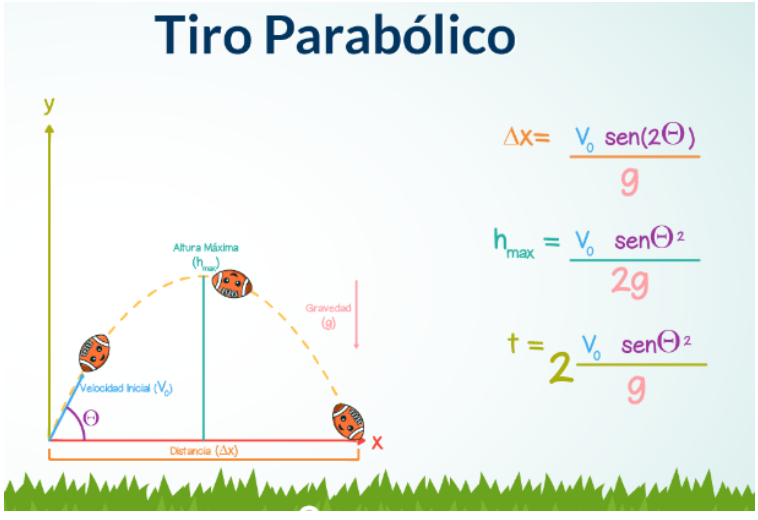
Definición del juego.

Rust in metal: El juego estará ambientado en un mundo donde el pj principal es un joven amante al metal y no tiene creencias en Dios ni en ningún ser divino, solo vive y existe para el mismo. Un día, tranquilo en su casa tocan la puerta unos evangélicos a predicar sus creencias, lo cual el les trata de tirar la puerta cuando uno de ellos pone el pie y lo evita, con ello el asustado con las acciones de los evangélicos corre a su habitación para buscar cómo defenderse de sus palabras y en el momento aparece la banda Tenacious D y le dan una super guitarra dorada hecha con metal puro y duro, con esta, donde apunte y rasga las cuerdas, convertirá en metalero al que sea que lo toquen las notas que salgan por su guitarra y esto con el fin de no dejar rastro de los evangélicos en el mapa. El juego se desarrollará en oleadas, cada una con mayor cantidad de enemigos y aumentando su velocidad de movimiento, la vida del jugador contará con 3 guitarras doradas y se eliminan al colisionar con un enemigo, una vez pierdas las tres guitarras el jugador se convertirá y dejará de ser metalero.

Físicas planteadas:

1. Saltos parabólicos: Con los saltos parabólicos se plantea hacer una habilidad especial del jugador donde lance un objeto y que va rebotando en línea recta para poder eliminar los enemigos de su alrededor, también se planea usar como uno de los movimientos de los enemigos para poder alcanzar al jugador y poder derrotarlo.

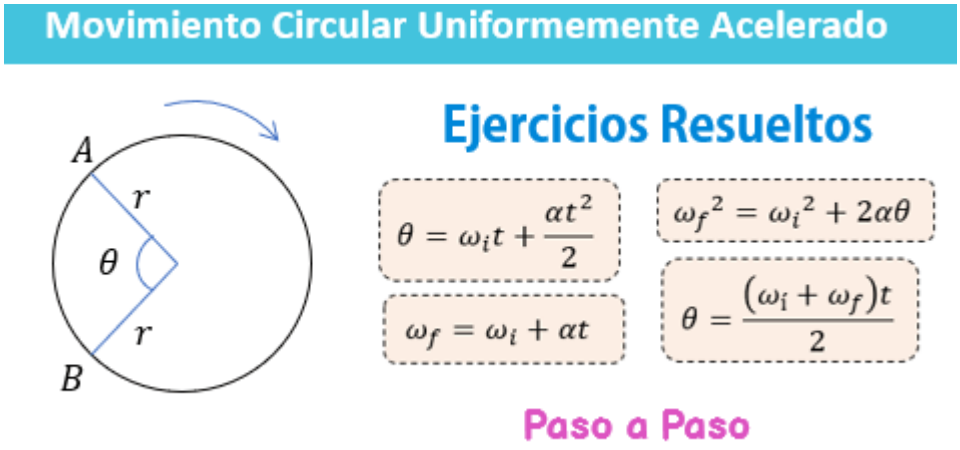


2. Salto normal(usar gravedad): Para los saltos simples se planea implementar en el apartado de los enemigos, como una opción de esquivar las notas que lanza el jugador y darles una segunda oportunidad a los enemigos de seguir en combate.

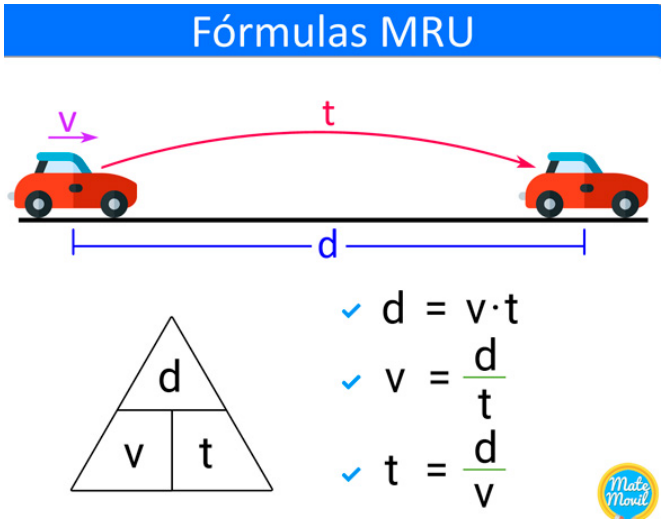
$$\begin{cases} V_f = V_0 - gT \\ Y = V_0 T - \frac{1}{2} g . T^2 \end{cases}$$

Con esta ecuación podemos aplicar el sistema para que los enemigos puedan saltar en un mismo punto, el movimiento se complementará con el rectilíneo uniforme.

3. Movimiento circular uniforme: Esto se usará como habilidad y como movimiento de los enemigos, los cuales tendrán unas esferas que girarán alrededor de ellos y será un especie de escudo protector o también se podrán lanzar(Toca analizar la posibilidad de implementarlo).



4. Movimiento rectilíneo uniforme: Esta física se usará para el movimiento de ciertos enemigos y para el jugador que se moverá en un mismo eje.



Procesos: La idea principal es iniciar con el desarrollo del mapeado y las mecánicas del juego, se procederá con iniciar aplicando las físicas y colisiones del personaje y los enemigos, con la idea de adelantar todos los movimientos base.
Desarrollar la generación del mapa y sus colisiones con el jugador y enemigos.
Desarrollar sprites específicos para ciertos enemigos si se desarrolla la base principal.

Clases a implementar:
Para poder desarrollar el juego correctamente, se requieren de ciertas clases que nos ayudarán a definir la forma de algunas acciones y objetos que tendrá el juego para desarrollar bien su planteamiento:

1. Clase jugador: En esta clase vamos a definir algunos atributos del jugador como lo son el movimiento, la vida y sus acciones(Movimientos de sprites).
   Con lo que se define su movimiento vertical, la tecla que usará para disparar las notas y la tecla que usará para lanzar habilidades, se realizará un contador de daño donde cada que reciba un golpe de los enemigos pierde un punto y podrá perder si llega a 0.
   El jugador heredará o traerá las funciones de física y disparo para poder ejecutar bien sus acciones.
   Para el jugador se usarán:

```h
#ifndef JUGADOR_H
#define JUGADOR_H

class jugador
{
private:
    int x;
    int y;

public:
    jugador();
    int getX() const;
    void setX(int value);
    int getY() const;
    void setY(int value);
    void posicion();
    void posicion(int newX, int newY);
    void MoveUp();
    void MoveDown();
    void Nota();
    void habilidad();

};


#endif // JUGADOR_H
```

2. Clase físicas: En esta clase construiremos las diferentes físicas a implementar definiendo sus ecuaciones, que variables hay que mantener constantes, que otras podrán variar dependiendo de los movimientos y también setear posiciones.

   Algunas de las físicas requerirán de las mismas variables que se muestran a continuación:

```h
#ifndef FISICAS_H
#define FISICAS_H

#define G 2;
class fisicas
{
private:
    float px;
    float py;
    const float rad;
    float vx;
    float vy;
    float ax;
    float ay;

public:
    fisicas();
    void bola(float px_=0, float py_=0, float vx_=0, float vy_=0, float rad_=10);

    float getPx() const;
    void setPx(float value);

    float getPy() const;
    void setPy(float value);

    float getVx() const;
    void setVx(float value);

    float getVy() const;
    void setVy(float value);

    float getAx() const;
    void setAx(float value);

    float getAy() const;
    void setAy(float value);

    float getRad() const;

    void mover(float dt);
};
#endif // FISICAS_H
```

3. Clase enemigo: Los enemigos se les proporcionará un modelo de movimientos predefinidos en la clase físicas y podrán moverse de acuerdo a la oleada, el tiempo transcurrido y un movimiento especial(movimiento parabólico, rectilíneo uniforme, etc), con lo que la meta de ellos es impactar contra la tarima del jugador y así provocar daño para que pierda el jugador, los enemigos tendrán sus habilidades también donde podrán disparar o lanzar cosas a la tarima para ocasionar eventos donde el jugador quede conmocionado o su movimiento sea más lento.
4. Clase disparo: Tanto jugador como enemigos podrán disparar, en el caso del jugador serán notas musicales con las que impactará contra los enemigos convirtiéndolos en metaleros, también se definirá los movimientos de las habilidades especiales las cuales se usarán para disparar mas rapido, disparar ondas expansivas o notas de

area(Notas que hacen daño en área), para los enemigos ellos lanzaran o dispararon objetos religiosos para evangelizar al jugador, su daño cambia dependiendo de la oleada y el jugador podrá destruir estos objetos con sus notas para evitar el daño.

```
#ifndef BULLET_H
#define BULLET_H

#include <QPainter>
#include <QGraphicsItem>



class bullet: public QGraphicsItem
{

private:
    int x;
    int y;
    int radio;

public:
    bullet(int rad);
    ~bullet();

    int getX() const;
    void setX(int newX);
    int getY() const;
    void setY(int newY);

    //Definir posicion graficar
    void posicion();
    void posicion(int newX, int newY);

    void shot(int);

};

#endif // BULLET_H
```

5. Clase muro: Esta clase nos permitirá no solo generar las colisiones y los muros que limitan el terreno, sino también agregar imagen y posiblemente sonido de ambiente al juego para darle la temática, la idea es que por oleadas cambien la música y su ritmo aumente a la vez que la velocidad de los enemigos, por otro lado, esta clase nos permitirá definir los límites de la colisión del jugador con los ejes, definir los límites de los enemigos y qué hacer cuando colisionan con una nota, o contra la tarima.

```
#ifndef MUROS_H
#define MUROS_H

#include <QGraphicsItem>
#include <QPainter>
#include <string>
#include <QPixmap>

using namespace std;

class Muros : public QGraphicsItem
{
    int posx, posy;
    int dimx, dimy;
    int MuroTipe;

public:
    Muros(int, int );
    Muros(int , int , int );
    virtual QRectF boundingRect() const;
    virtual void paint(
        QPainter *painter,
        const QStyleOptionGraphicsItem *option,
        QWidget *widget = nullptr);
    int getMuroTipe() const;
    void setMuroTipe(int newMuroTipe);
};

#endif // MUROS_H
```

6. Clase mainwindow: Esta clase nos permitirá invocar los enemigos, muros y jugador para diseñar el mapa y la jugabilidad, también se le especificará los destructores, cuando eliminar cosas y ventanas emergentes para decir si ganó, perdió o inicio el juego al usuario.

posibles gráficas: