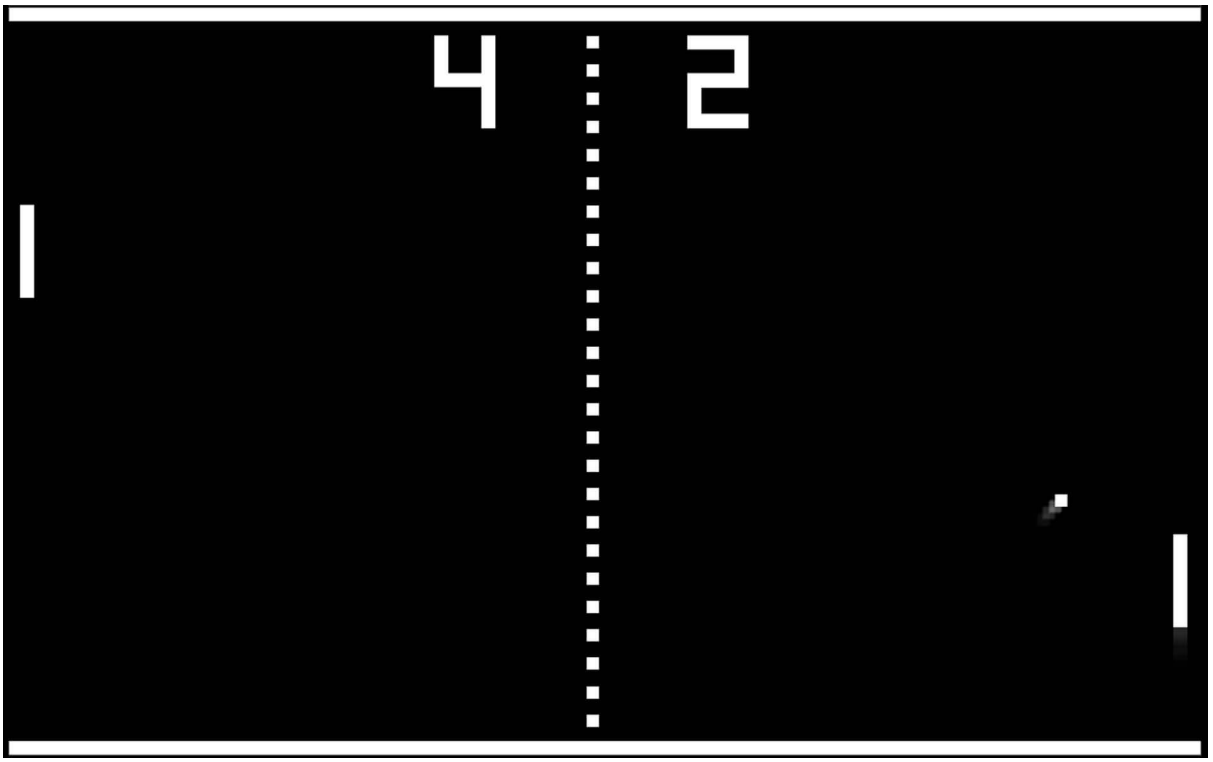
Pong co-op

Gabriel Ribeiro Fonseca de Freitas - 12542651
Gabriel Barbosa de Oliveira - 12543415

Introdução

O **Pong co-op** é uma releitura do famoso e antigo Pong Game do Atari.

Utilizamos como template para nosso projeto o seguinte repositório: Template. A licença e os devidos créditos estão disponíveis no repositório do projeto.



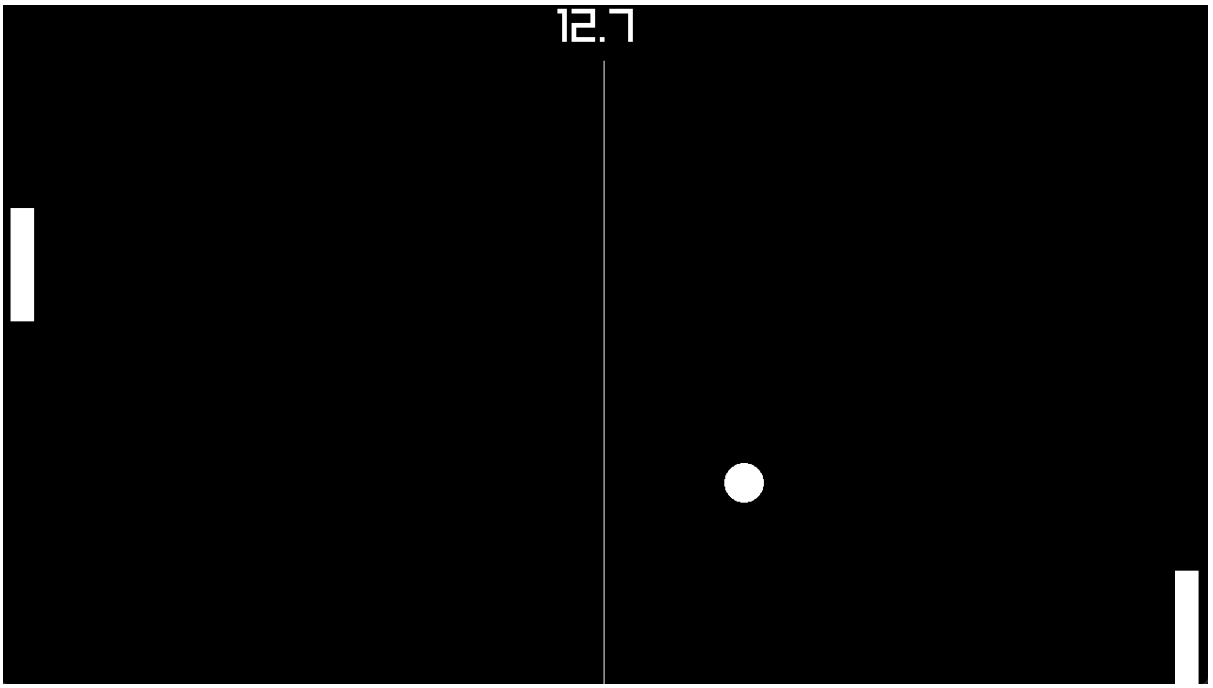
Pong Game- Atari 1972

Regras

Diferentemente do original, as duas raquetes não podem se mover simultaneamente e o jogo acaba quando a bola bate em alguma das laterais da tela. Ou seja, é necessária a coordenação do(s) player(s) para manter a bola em jogo.

A cada “raquetada”, a bola e a raquete aceleram um pouco. Ou seja, conforme o tempo passa, o jogo fica mais frenético e é cada vez mais difícil rebater.

O objetivo principal é conseguir manter a bola em movimento e a pontuação é o tempo que demorou até o jogo terminar.



Jogo em execução

Controles

Aperte as teclas **I** e **J** para mover a raquete da direita para cima e para baixo. A raquete da esquerda é controlada com **W** e **S**.

Instalação e Execução

É necessário ter o programa **Make** instalado para compilar o programa.

Para instalar o jogo, basta clonar o seguinte repositório do GitHub: PongGame.

Para rodar, entre no diretório pelo terminal, compile e execute o jogo com os seguintes comandos:

```
make
./game.exe
```

Tela inicial do jogo

Código

Primeiramente, vamos dar um panorama geral das ferramentas utilizadas na confecção do jogo:

- Linguagem: C++;
- Interface gráfica e funcionalidades do jogo: Raylib (biblioteca para o desenvolvimento de jogos de forma simples e prática);
- Bibliotecas adicionais:
    - <thread> : Utilização de Threads;
    - <mutex> : Utilização de Semáforos;
    - <chrono> : Medição de tempo;
    - <iostream> : Tratamento de operações de entrada / saída;

```
#include <thread>
#include <iostream>
#include <mutex>
#include <chrono>
#include "raylib/raylib/src/raylib.h"
```

Exemplo de includes

O código foi modularizado, separando as classes e variáveis de forma organizada em arquivos separados:

```
main.cpp
globals.cpp
global.h
Ball.cpp
Ball.h
Paddle.cpp
Paddle.h
```

De maneira resumida, o jogo consiste em um loop que executa até que a janela seja fechada, ou até que o player aperte a tecla Q nos menus onde essa opção está presente. Cada iteração do loop representa um frame. Dessa forma, cada classe, mais especificamente as classes *Paddle* (raquete) e *Ball* (bola), possuem suas respectivas funções *Update()*, que atualiza a posição do objeto a cada frame da iteração.

```
void Ball::Update()
{
    if (currentScreen == GAMEPLAY)
    {

        x += speed_x;
        y += speed_y;

        if (y + radius >= GetScreenHeight() || y - radius <= 0)
        {
            speed_y *= -1;
        }
        if (x + radius >= GetScreenWidth() || x - radius <= 0)
        {
            GameOver();
        }
    }
}
```

Exemplo de função Update()

Cada raquete é uma instância da classe *Paddle*. Cada uma dessas instâncias, tem a execução de sua respectiva função *Update()* em uma *Thread*:

```
    thread thP1(update_player_in_thread, &player1);
    thread thP2(update_player_in_thread, &player2);
```

```
void update_player_in_thread(Paddle *player)
{
    while (WindowShouldClose() == false)
```

```
    {
        mtx.lock();
        (*player).Update();
        mtx.unlock();
    }
}
```

Assim, temos, a cada iteração do loop principal, as duas threads, ou seja, as duas funções *Update()*, são executadas simultaneamente. Então, para sincronizar a utilização dos recursos desse recurso de entrada, utilizamos *mutex* (mtx), como ilustrado no código acima.

Além disso, para garantir que não seja possível que as duas raquetes se movam simultâneamente, a thread é suspensa durante um curto período de tempo:

```
    // move up
    if ((wasd && IsKeyDown(KEY_W)) || (!wasd && IsKeyDown(KEY_I)))
    {
        y -= speed;
        this_thread::sleep_for(chrono::milliseconds(5));
    }

    // move down
    else if ((wasd && IsKeyDown(KEY_S)) || (!wasd && IsKeyDown(KEY_K)))
    {
        y += speed;
        this_thread::sleep_for(chrono::milliseconds(5));
    }
```

Da documentação da função *sleep_for*: "*Blocks the execution of the current thread for at least the specified sleep_duration*."

Assim, a thread do outro player não poderá ser executada durante esse tempo, pois a função *unlock()* do *mutex* não será invocada.