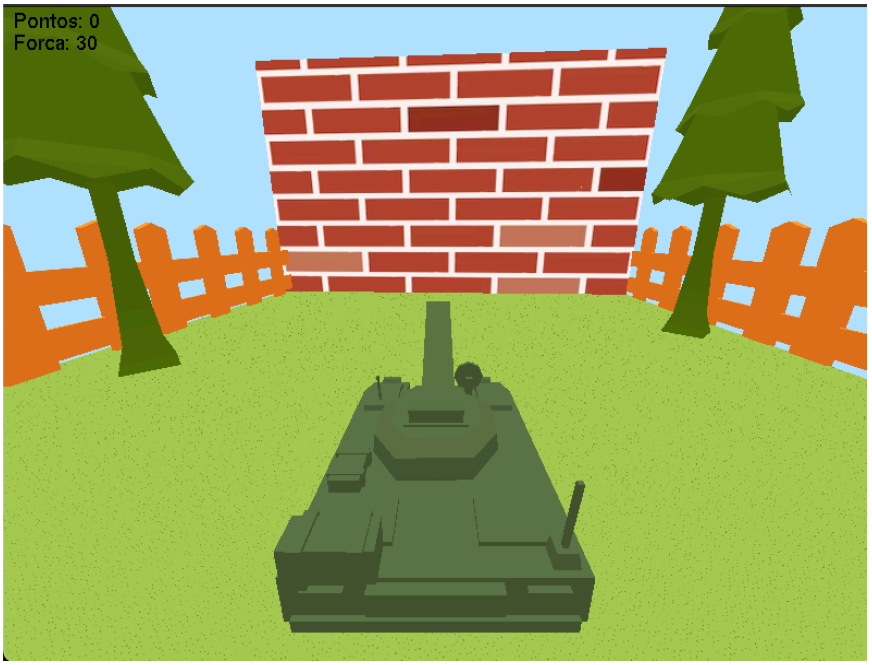
Trabalho 2 Computação Gráfica

Francisco borba

Desenho do Paredão

O desenho deve ser feito com cada célula sendo gerada individualmente, com uma textura aplicada a cada célula. A textura deve vir da mesma imagem, mas em cada célula, uma parte diferente da imagem deve ser mapeada.



Para atender tal requisito foi utilizada uma textura *brick.png* que está sendo aplicado em uma classe *Paredão* que tem o seguinte construtor:

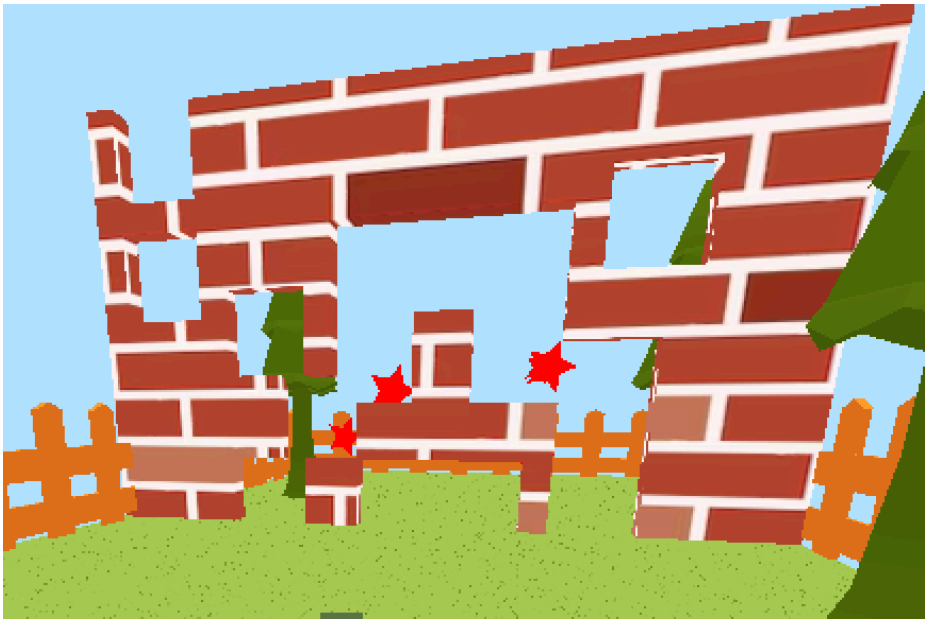
```
/**
 * @brief Construtor da classe Paredao.
 *
 * @param largura Largura do paredão (em cubos 1x1x1).
 * @param altura Altura do paredão (em cubos 1x1x1).
 * @param posicao Posição do canto inferior esquerdo do paredão.
 * @param seed Semente para a geração de cores.
 */
Paredao::Paredao(int largura, int altura, Ponto posicao, int seed)
    : largura(largura), altura(altura), posicao(posicao), seed(seed)
{
  cubos.resize(largura, std::vector<Cubo>(altura)); // Inicializa os cubos
}
```

Para criar o paredão com a textura temos os métodos:

- carregarTextura(const char *caminho);
- voiid desenhaParedao();
- void desenhaCubo(int corBorda, float texCoordX, float texCoordY, float texWidth, float texHeight);

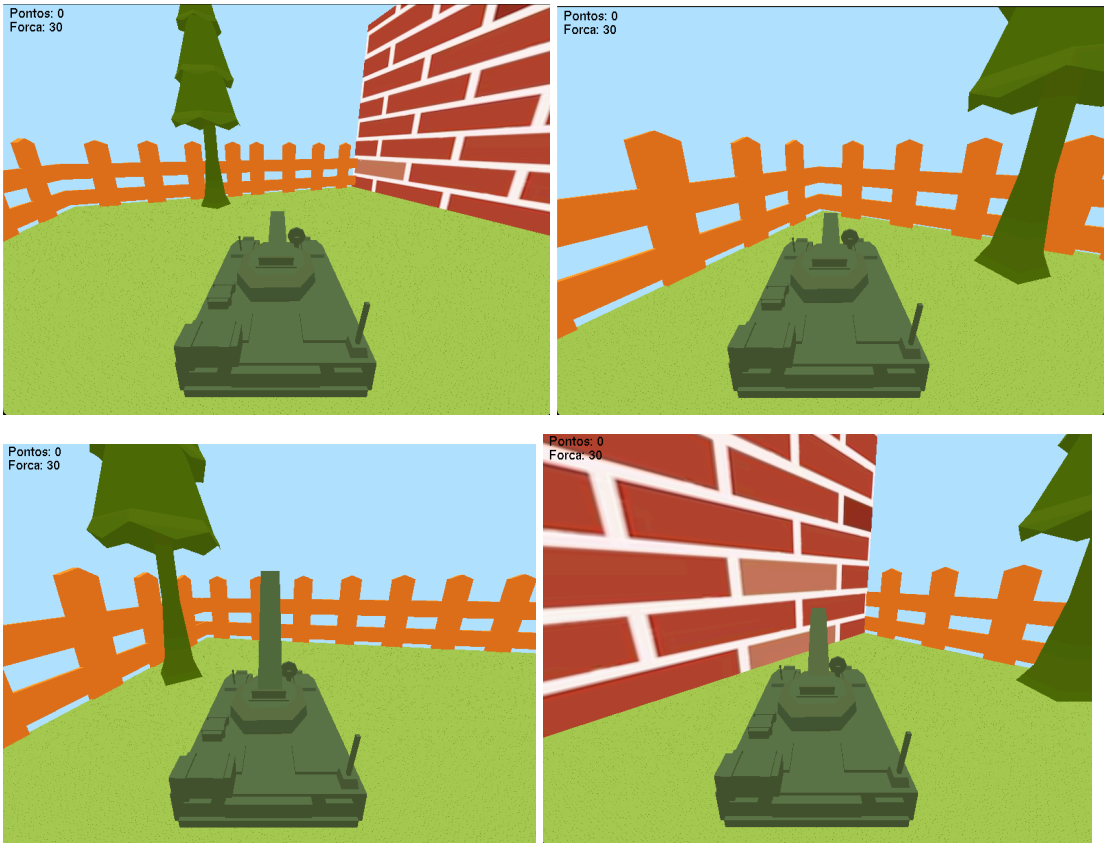
Que lidam com toda esta parte de dividir a textura para cada bloco ter uma parte da mesma.

Detecção da colisão do projétil com o paredão e com os objetos do cenário e reconfiguração do Paredão após a colisão com o projétil



A função *detectarColisao(Ponto objetoPosicao)* é a responsável por lidar com colisões e verifica se o ponto recebido colide com algum cubo do paredão. Em caso de colisão, o cubo atingido e seus 8 vizinhos em um raio de 1 unidade (formando um quadrado 3x3) têm a propriedade *quebrado* marcada como *true*, removendo-os do cenário. Assim, a colisão com o tiro não só é detectada como também reconfigura visualmente o paredão, formando um buraco maior na área atingida.

Modelagem do Veículo com as articulações móveis e Movimentação do Veículo - girar e andar para frente



As imagens acima mostram o jogador em diversos lugares do ambiente. O que demonstra a sua movimentação. Essa movimentação do jogador é feita em conjunto com a movimentação da câmera e segue estas duas funções:

```
// ************************************************************************
//  void Camera::goForward(int direction)
//  Move a câmera para frente ou para trás baseado na direção fornecida.
//
//  Parâmetros:
//      int direction: Direção do movimento (0 para frente, 1 para trás).
// ************************************************************************
void Camera::goForward(int direction)
{
  float walkSpeed = direction == 0 ? 0.15f : -0.15f;

  // Normaliza o vetor direção (vetor de alvo)
  Ponto VetorAlvoUnitario = *VetorAlvo;
  VetorAlvoUnitario.versor(); // Normaliza o vetor direção

  // Calcula o movimento
  VetorAlvoUnitario.multiplica(walkSpeed, walkSpeed, walkSpeed);

  // Nova posição de OBS e ALVO após o movimento
  Ponto novoOBS = *OBS;
  novoOBS.soma(VetorAlvoUnitario.x, 0, VetorAlvoUnitario.z);

  Ponto novoALVO = *ALVO;
  novoALVO.soma(VetorAlvoUnitario.x, 0, VetorAlvoUnitario.z);

  // Verificação se o novo OBS e ALVO estão dentro dos limites
  if (novoOBS.x + novoALVO.x * 0.38f >= -15.0f && novoOBS.x + novoALVO.x * 0.38f <= 15.0f &&
      novoOBS.z + novoALVO.z * 0.38f >= 2.0f && novoOBS.z + novoALVO.z * 0.38f <= 30.0f)
  {
    // Se dentro dos limites, aplica o movimento
    *OBS = novoOBS;
    *ALVO = novoALVO;

    player->setOBS(*OBS);
    player->setVetorAlvo(*VetorAlvo);
  }
  // Caso contrário, a câmera não move.
}
```

```
// ************************************************************************
//  void Camera::lookSideways(int direction)
//  Faz a câmera olhar para os lados baseado na direção fornecida.
//
//  Parâmetros:
//      int direction: Direção do movimento (0 para esquerda, 1 para direita).
// ************************************************************************
void Camera::lookSideways(int direction)
{
  ALVO->x -= OBS->x;
  ALVO->z -= OBS->z;

  float alfa = (direction == 0) ? 0.02 : -0.02; // Define o ângulo de rotação
  float x = ALVO->x * cos(alfa) + ALVO->z * sin(alfa);
  float z = -ALVO->x * sin(alfa) + ALVO->z * cos(alfa);
  ALVO->x = x + OBS->x;
  ALVO->z = z + OBS->z;

  VetorAlvo->x = ALVO->x - OBS->x;
  VetorAlvo->y = ALVO->y - OBS->y;
  VetorAlvo->z = ALVO->z - OBS->z;

  player->setOBS(*OBS);
  player->setVetorAlvo(*VetorAlvo);
}
```

Elas são responsáveis por movimentar o veículo para frente e para trás além de é claro rotacionar ele permitindo movimentos para o lado e diagonais. Isso tudo é controlado na função *updateCamera()* em conjunto com uma classe *KeyboardController* que trabalha com um vetor de booleanos que representam os estados de cada uma das teclas. Garantindo assim um movimento fluido.

```
void Camera::updateCamera()      You, 3 weeks ago • Camera
{
  if (keyboard.isKeyPressed('w'))
  {
    goForward(0); // Move para frente
  }
  if (keyboard.isKeyPressed('s'))
  {
    printf("s");
    goForward(1); // Move para trás
  }
  if (keyboard.isKeyPressed('a'))
  {
    lookSideways(0); // Olha para a esquerda
  }
  if (keyboard.isKeyPressed('d'))
  {
    lookSideways(1); // Olha para a direita
  }
  if (keyboard.isKeyPressed('q'))
  {
    player->raiseCannon(1); // Levanta o canhão
  }
  if (keyboard.isKeyPressed('e'))
  {
    player->lowerCannon(1); // Abaixa o canhão
  }
  // se a tecla for flecha para cima printar "oi"
  if (keyboard.isKeyPressed('['))
  {
    PontosManager::diminuirVelocidadeTiro(1);
  }
  if (keyboard.isKeyPressed(']'))
  {
    PontosManager::adicionarVelocidadeTiro(1);
  }
```

Lançamento do Projétil, com base no ângulo do canhão e do veículo

A classe *Tiro* foi criada para lançar projéteis, recebendo uma posição de saída, um alvo e uma velocidade. O movimento segue uma parábola simulando gravidade, com a posição atualizada pela função *updateTiro*(). A origem e o alvo são baseados na classe *Player*, utilizando sua posição como origem e a propriedade *cannonAngle* para calcular o alvo. Segue abaixo a classe utilizada pelo *Player* para realizar o disparo:

```
void Player::dispararTiro(Ponto cameraAlvo)
{
  printf("Disparando tiro\n");
  // Vetor direção baseado no vetor alvo
  Ponto direction = cameraAlvo;

  // Usa o ângulo do canhão para calcular a direção.y
  direction.y = tan(cannonAngle * M_PI / 180.0f) * 5.0f;

  // Cria um tiro na posição do jogador e na direção do vetor alvo
  // int speed = PontosManager::getVelocidadeTiro();
  // seed tem que ser igual a getVelocidadeTiro dividido por 100 e no formato float
  float speed = PontosManager::getVelocidadeTiro() / 100.0f;
  Tiro tiro(position, direction, speed, speed);

  // Adiciona o tiro ao vetor de tiros
  tiros.push_back(tiro);

  // Se o número de tiros for maior que 10, remove o mais antigo
  if (tiros.size() > 10)
  {
    tiros.erase(tiros.begin()); // Remove o primeiro elemento (o mais antigo)
  }
}
```

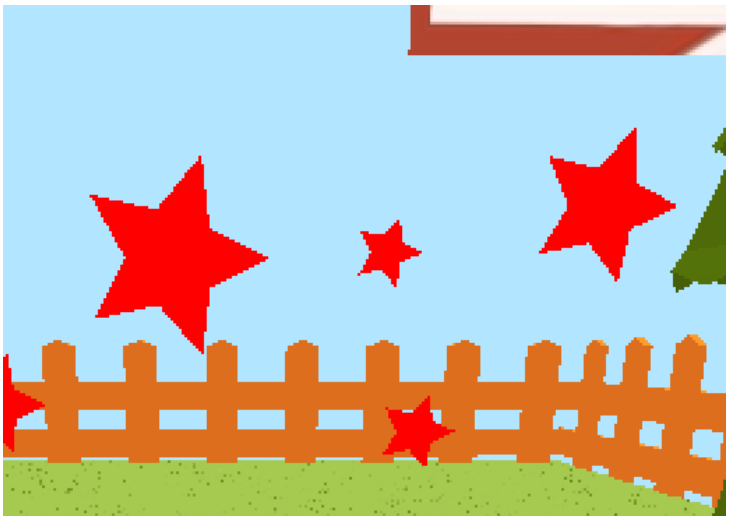
Exibição correta dos objetos 3D

O cenário é composto por diversos objetos, como árvores que preenchem o ambiente, cercas que delimitam a zona de jogo e estrelas, que funcionam como alvos. Para gerenciar esses elementos, foi criada a classe *Modelo3D*, permitindo que objetos 3D sejam utilizados de forma padronizada. Dentro desta classe tem toda lógica para carregar modelos.

```
arvore = new Modelo3D(10.0f, -1.0f, 10.0f);
arvore->carregarModelo("./assets/models/tree.obj");
arvore->setColor(0.17f, 0.23f, 0.02f);
arvore->setEscala(8.0f, 8.0f, 8.0f);
arvore->setRotacao(0.0f, 45.0f, 0.0f);
modelos.push_back(*arvore);
```

Fora isso, o modelo do jogador também foi substituído e agora é um modelo de tanque. Entretanto o cano do tanque segue sendo um paralelepípedo.

Segue abaixo algumas imagens dos modelos utilizados no projeto:



Link para o vídeo : https://www.youtube.com/watch?v=Qf1xV5zh-JQ