WILLIAN AYRES WEB DEVELOPER
RELATÓRIO DO PROJETO
SIMULADOR DE SNAKE

Willian Joris Ayres

RELATÓRIO FINAL DO PROJETO

SOFTWARE PARA SIMULADOR DE SNAKE

Distração e entretenimento

Curitiba

Fevereiro / 2020

Willian Joris Ayres

RELATÓRIO FINAL DO PROJETO

SOFTWARE PARA SIMULADOR DE SNAKE

Distração e entretenimento

Relatório final de projeto, apresentado como requisito de confirmação de conhecimentos no desenvolvimento de softwares em linguagem de programação C/C++.

Curitiba

Fevereiro / 2020

Sumário

Lista de ilustrações

Lista de apêndices

Introdução

Este documento apresenta a versão final do simulador do jogo “Snake” feito por Willian Ayres. Tendo em vista como aperfeiçoamento de conhecimentos prévios de lógica de programação, algoritmos, desenvolvimento de softwares na linguagem C e C++, além da melhoria na abstração de resolução de problemas, tanto pelo meio procedural, quanto pela orientação à objetos.

O contexto de codificação deu-se pela IDE “Code Blocks”, sendo de fácil acesso e gratuito. Existem vários motivos para utilização dessa IDE, por exemplo, ela possui várias ferramentas adicionais, como os “debuggers” e os “compilers”.

O código, em sua integridade, é escrito completamente em C++, sem muitas enrolações. É compreensível e conciso para quem tem uma mínima noção de lógica, conhecimentos da linguagem C++ e abstração. Algumas de suas partes poderiam ser mais otimizadas, porém foram deixadas brechas para possíveis futuros ajustes.

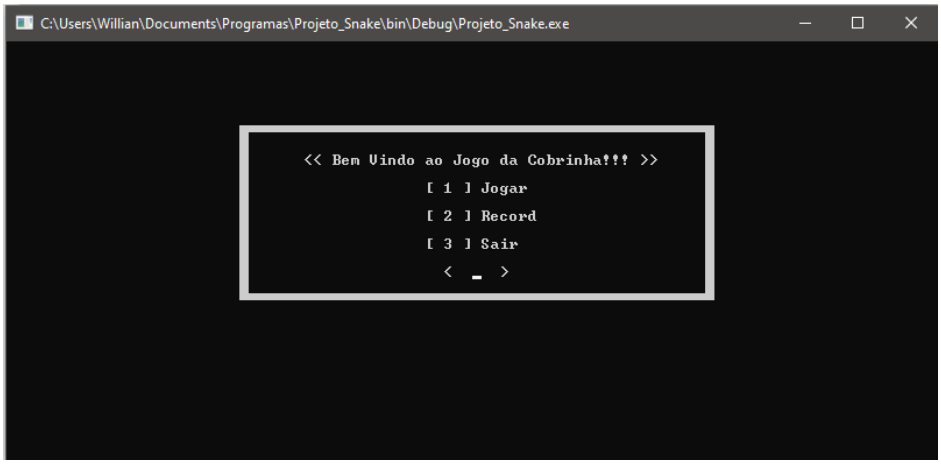
Se você é atraído pelas belezas do mundo da programação, seguir esse relatório para criar o seu próprio simulador de Snake não será um grande problema. Basta seguir corretamente e ajustar conforme suas métricas e seus próprios gostos.

Explicação do simulador

O simulador de Snake inicia-se com a impressão do menu no console (como qualquer jogo padrão).

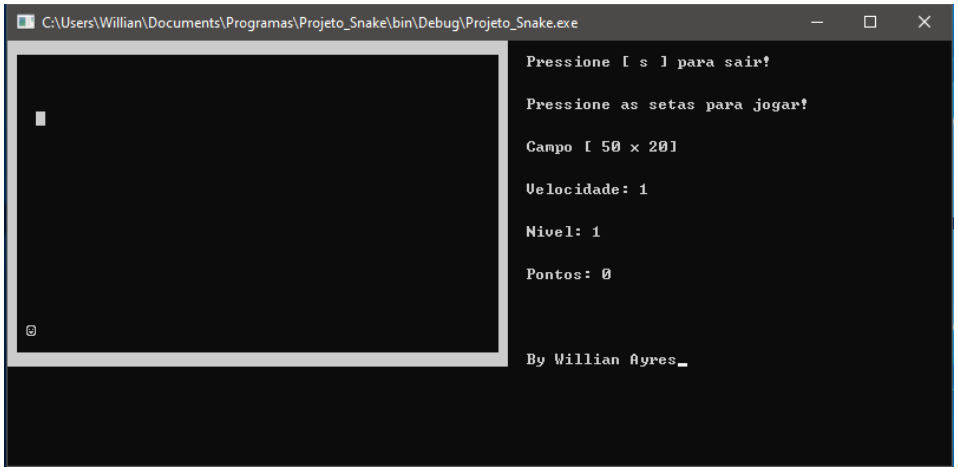
Exibe-se uma saudação ao usuário junto do nome do simulador. Logo após, às 3 opções de escolha possíveis: Jogar, Record ou Sair. Espera-se a escolha do usuário:

Menu:



Caso a escolha do jogador seja ‘1’, então o jogo é iniciado.

Início do jogo:

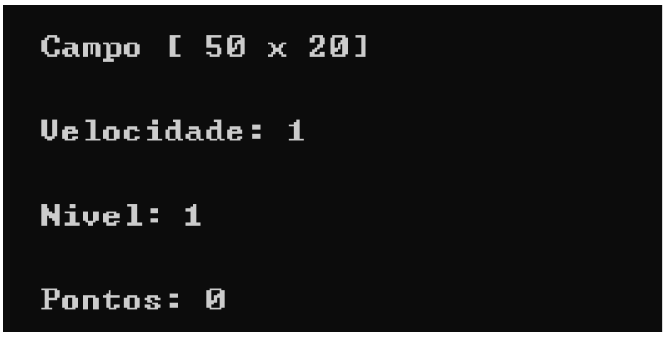


Todas as informações necessárias para o decorrer do jogo, estão presentes ao lado do campo. Dentre elas, as primeiras são referentes as entradas de tecla do usuário. Caso ele queira desistir, ou parar em algum momento, deve-se apenas apertar a tecla ‘s’ que o jogo se encarregará de terminar naquele instante.

Para movimentar a “cobrinha” dentro do campo, o usuário deve apertar as teclas de seta do teclado, fazendo com que o movimento dela seja na direção da tecla pressionada. Por exemplo, ela está indo para a direita horizontalmente e então, o usuário pressiona a tecla para cima. Automaticamente ela iniciará o movimento para cima.

As próximas informações apresentas são referentes ao jogo em si. A primeira indica as dimensões do campo jogado (nesse simulador é sempre a mesma, mas pode ser facilmente modificada conforme gosto do desenvolvedor. A velocidade, como no nome, indica a velocidade com que a “cobrinha” movimenta-se e é incrementada se uma maça é pega. Quanto maior a velocidade, mais rápido o jogo será executado. O nível é apenas um indicador de dificuldade, informando ao usuário que quanto mais maças são pegas, maior será a dificuldade por conta do aumento da velocidade. Os pontos apenas indicam a quantidade de maças pegas, que particularmente, indica o tamanho da “cobra”.

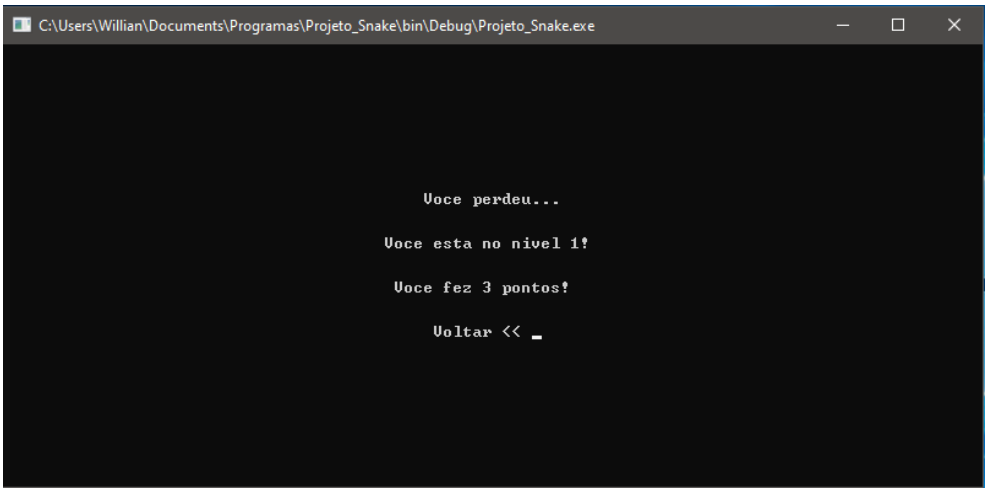
Informações do jogo:



Jogo acontecendo:

Caso o jogador erre (bata nas paredes ou bata na própria “cobrinha”) ou aperta a tecla de escape ‘s’, o jogo termina e são exibidas algumas informações finais na tela:

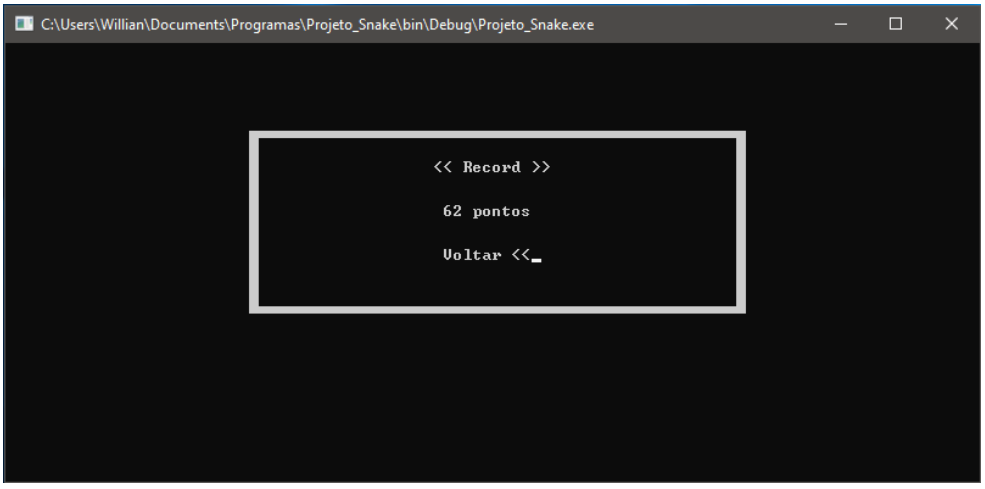
Final do jogo:



Informa que o usuário perdeu, qual nível alcanço e sua pontuação final. Então espera qualquer entrada para retornar ao menu principal.

No menu principal, se a escolha seja ‘2’ (Records), é exibido a maior pontuação, até o momento. Essa pontuação será atualizada caso algum usuário supere a mesma.

Exibição do record:



E por último, caso o usuário escolha a opção ‘3’ (Sair), o programa é encerrado.

Codificação do Simulador

Para a estruturação do código em geral, foram usadas as bibliotecas: Windows.h para posicionar o cursor no console, conio.h e iostream para interação com o usuário. Para leitura de arquivos texto: a biblioteca fstream. Para armazenamento de palavras e arrays: as bibliotecas: string e vector. Além dos arquivos cabeçalhos definidos pelo desenvolvedor.

Para auxilio de posicionamento do cursor no console, implementa-se uma função com a ajuda da Windows.h no qual os parâmetros x e y serão as novas coordenadas para o cursor no console:

```
void nome_genérico (short x, short y)
{
 SetConsoleCursorPosition (GetStdHandle (STD_OUTPUT_HANDLE), (COORD) {x, y});
}
```

```
void menu () {
   int c;
   do {
      print_menu ();
      c = getchar ();
   } while (c != 49 && c != 50 && c != 51);
   switch(c) {
      case 49: game (); menu (); break;
      case 50: print_record (); break;
      case 51: break;
   }
}
```

O simulador do jogo “Snake” inicia-se com apenas uma chamada para uma função genérica menu (). Pode-se imprimir as opções no menu diretamente na tela, ou também separar em alguma outra função. As impressões ficam a critério do desenvolvedor, como sua posição no console, maneira como é impresso, caracteres especiais ou até mesmo o básico. Como a entrada é por meio do getchar, a variável c possuirá um valor unsigned int, e o caractere será aquele da tabela ASCII. No caso, 49 corresponde ao número 1, e assim por diante. Essa variável continuará sendo requisitada do teclado enquanto for diferente de 1, 2 ou 3. Então, a variável c é colocado no switch e caso seja 1, será chamada uma para iniciar função o jogo e após uma recursiva para o menu. Caso 2, a função para mostrar o Record. Caso 3, o programa será encerrado.

Antes da explicação sobre o funcionamento da função game, precisa-se introduzir algumas classes:

```
class Campo
{
   private:
      int tamx;
      int tamy;
      std::string nome;
   public:
      Campo ();
      int get_tamx ();
      int get_tamy ();
      std::string get_nome ();
      void print_campo ();
};
```

```
class Cobra
{
   private:
      int t;
      int d;
   public:
      Cobra ();
      std::vector <int> cx;
      std::vector <int> cy;
      void set_cx (int);
      void set_cy (int);
       int get_t ();
       void att_t (int);
       int get_d ();
       void att_d (int);
};
```

```
class Info
{
   private:
      int vel;
      int vel2;
      int pontos;
      int nível;
   public:
      Info ();
      int get_vel ();
      void att_vel (int);
      int get_vel2 ();
      void att_vel2 (int);
      int get_pontos ();
      void att_pontos (int);
      int get_nivel ();
      void att_nivel ();
};
```

```
class Maca
{
   private:
      int mx;
      int my;
   public:
      int get_mx ();
      int get_my ();
      void new_maca (int, int);
};
```

Para a classe Campo, precisa-se de uma altura e um comprimento, além de um possível nome. Métodos de impressão dos campos e do nome são opcionais. Métodos de acesso as variáveis são necessários. O método construtor deverá ser personalizado para inicialização das variáveis conforme métrica do desenvolvedor.

Para a classe Cobra, precisa-se de um vetor de coordenadas para x e para y, nas quais cada par de coordenadas refere-se à um pedaço da cobra. Precisa-se também do tamanho da cobra e da direção do movimento dela. Métodos de atualização e recuperação das variáveis e empilhamento de novas coordenadas são necessários.

Para a classe Info, precisa-se de variáveis para armazenar as informações do jogo, por exemplo velocidade da cobra, do jogo, número e pontos e o nível. Além dos métodos de atualização e recuperação dessas variáveis.

Para a classe Maca, precisa-se das coordenas x e y da maça. São necessários os métodos de recuperação dessas variáveis e um método para gerar novas coordenadas.

No início da função game, são instanciados novos objetos para cada uma dessas classes, impresso as bordas do campo e gerada uma nova maçã no campo.

```
void Campo::print_campo ()
{
   For (int i = 0; i < this -> tamy; i++)
   {
      goto_XY (0, i);
      std::cout << (char)219;
   }
}
```

Repetir para cada aresta do campo.

O caractere para as bordas é arbitrário.

```
void Maca::new_maca(int x, int y)
{
   srand (time (NULL));
   this->mx = (rand () % (x-1)) + 1;
   this->my = (rand () % (y-1)) + 1;
}
```

Gera uma maça aleatória com x e y entre 1 e as dimensões do campo.

O jogo inteiro passa-se dentro de loop while que é quebrado apenas se a tecla de escape for igual a ‘s’. Então, usa-se outro laço para leitura das teclas usadas para movimento do jogador. Para a leitura da condição de tecla pressionada, usa-se a função kbhit () da biblioteca conio.h, junto do operador &&! na condição do laço while.

```
while (saida != 's' &&! (saida = kbhit ()))
```

Algumas métricas são definidas para o loop ser quebrado: Se o usuário pressionar a tecla s; se a cobra colidir com ela mesma; se a cobra colidir com alguma das paredes.

```
for (int i = 1; i <= cobra.get_t (); i++) {
   if (cobra.cx [0] == cobra.cx [i] && cobra.cy [0] == cobra.cy [i]) {
      saida = 's'; break;}
}
if (cobra.cy [0] == 0 || cobra.cy [0] == campo.get_tamy () || cobra.cx [0] == 0 || cobra.cx [0] == campo.get_tamx ())
      saida = 's';
```

Para conferir alguma colisão entre os pedaços da cobra basta fazer o laço começando do segundo pedaço até o último, e conferir se as coordenadas desse pedaço coincidem com o primeiro pedaço da cobra. Se colidirem, o jogo é encerrado.

Para conferir alguma colisão entre a cobra e as bordas do campo, basta conferir se as coordenadas do primeiro pedaço da cobra coincidem com as coordenadas de extremidade do campo. Se colidirem, o jogo é encerrado.

```
for (int i = cobra.t; i > 0; i--)
{
        cobra.cx [i] = cobra.cx [i - 1];
        cobra.cy [i] = cobra.cy [i - 1];
}
```

Para fazer o movimento da cobra usa-se um loop que se encerra no tamanho da cobra. A cobra na posição anterior irá receber as coordenadas da posição atual, conforme o movimento dela. Essa coordenada será atualizada conforme direção do movimento.

```
switch (cobra.get_d ()) {
   case 0: cobra.cx [0] --; break;
   case 1: cobra.cy [0] --; break;
   case 2: cobra.cx [0] ++; break;
   case 3: cobra.cy [0] ++; break;
}
```

Para conferir a direção que a cobra irá percorrer, basta analisar o valor d do objeto cobra e colocá-lo em um switch. Os valores são arbitrários, porém cada valor deve conter uma atualização diferente. Para cobra ir para cima, sua variável em y deve reduzir. Para ir para direita, sua variável em x deve aumentar. Atualiza as quatro direções dessa maneira.

É preciso imprimir a cobra na tela posicionando o cursor na posição das coordenadas inicias da cobra e a maça nas posições dela no campo. Além disso, precisa-se usar um macete para limpar o rastro desnecessário da cobra, fazendo com que o campo na última posição dos vetores de coordenadas imprima um espaço.

```
goto_XY (cobra.cx [cobra.get_t ()], cobra.cy [cobra.get_t ()]);
std::cout << " ";
```

```
for (int i = 0; i < cobra.get_t (); i++) {
    if (cobra.cx [i] == maca.get_mx () && cobra.cy [i]
        == maca.get_my ())
    {
        maca.new_maca (campo.get_tamx (), campo.get_tamy ());
        i = 0;
    }
    else
        continue;
```

Caso a cobra encontre a maça no campo, ou seja, exista colisão entre a maça e a posição inicial do vetor de coordenadas da cobrinha, deve-se atualizar todas as informações conforme métrica do desenvolvedor. Assim que atualizadas as informações, gera-se uma nova maça no campo, porém ela não pode ser gerada sobre a cobra, então utiliza-se um laço para que isso não ocorra.

Após isso, conforme métrica do desenvolvedor, imprime-se as informações de pontuação e velocidade para o usuário. Então, é preciso conferir a velocidade de impressão de todas essas informações no console, por meio da função Sleep (). Utiliza-se como argumento a informação de velocidade na qual é incrementada com o aumento da pontuação.

```
Sleep (info.get_vel ());
```

```
if (saida != 's')
    saida = getch ();
if (saida == 'K')
    cobra.att_d (0);
if (saida == 'H')
    cobra.att_d (1);
if (saida == 'M')
    cobra.att_d (2);
if (saida == 'P')
    cobra.att_d (3);
```

Na leitura do teclado, utiliza-se outro macete, recuperando os valores das setas correspondentes na tabela ASCII. Dependendo da seta teclada, a direção será diferente. Se a tecla pressionada for diferente de ‘s’, recupera-se o valor. Apenas as setas serão consideradas.

```
void record (int pontos) {
   std::ifstream input("record.txt");
   if (! input.is_open ()) {std::cout << "Erro ao abrir o arquivo! \n";
      return;}
   int record; input >> record;
   std::ofstream output("record.txt");
   if (! output.is_open ()) {std::cout << "Erro ao abrir o arquivo! \n";
      return;}
   if (pontos > record) output << pontos;
   else output << record;
}
```

Após a quebra do laço while, limpa-se a tela do console, imprime-se as informações gerais do jogo que acabou e chama-se uma função para comparação de pontuação, na qual é passada o número de pontos como argumento. Nela, abre-se o arquivo texto com o record salvo, e recupera o valor nele escrito. Então, compara-se o valor recebido como parâmetro com o recuperado do arquivo e então, escreve no arquivo texto aquele que possuir o maior valor, tornando o valor salvo no arquivo texto como record.

Após conferido a pontuação, utiliza-se ou um system (“pause) ou getch () para espera de entrada aleatória do usuário, então, retornado ao menu.

Considerações Pessoais

Basicamente, esse projeto foi realizado no intuito de desenvolver melhores capacidades como programador em C++. Dizer da boca pra fora que sabe programar em alguma linguagem pode até ser fácil, mas colocar a mão na massa e produzir algo com aquilo que tem conhecimento pode ser uma tarefa complicada.

Nas primeiras etapas do desenvolvimento, a procura de funções para trabalhar com o console foram essenciais. Funções como posicionamento do cursor e tempo de espera pra impressão foram as primeiras a serem cogitadas na implementação do game. Após trilhar esse caminho, evidenciou-se necessário a utilização de classes e objetos para mostrar a parte de orientação ao objeto da linguagem. Com o esboço das classes prontas, a parte mais impactante e complexa mostrou sua faceta: a lógica do jogo. Como não é algo tão complexo, a lógica dele é relativamente "fácil", tanto para codificação, quanto para abstração. A lógica do movimento da cobra e do aumento ao adquirir pontos não foi o problema. Descobrir como fazer que uma maça fosse gerada aleatoriamente, sem ter colisões com o campo e com a cobra, se mostrou como tarefa mais complicada. Talvez pela lógica, alguns computadores com menor capacidade de processamento tenham dificuldade para executar a tarefa.

Quanto ao quesito leitura de arquivos texto, não é uma tarefa complicada, quando você precisa ler apenas uma informação e atualizar a mesma. Basta saber o momento certo de abrir, a maneira que irá abrir esse arquivo e executar as comparações de informação, para não ocorrer desperdício de memória.

Em geral, foram usados muitos dos conceitos da linguagem como: estruturas de repetição, estruturas condicionais, funções, datatypes, um pouco de strings e arrays, vetores, arquivos texto, overloading de métodos, classes e objetos, gettings e setters, além de inúmeros recursos dessa vasta linguagem de programação. O uso desses recursos depende muito do desenvolvedor, por ter mais compatibilidade e afinidade com algum em especifico. No caso desse simulador, a preferência ao uso de vetores ao invés de arrays dinâmicos ou estáticos, mostrou-se presente.

Enfim, para desenvolver um simulador como este, basta seguir o raciocínio lógico correto e ter em mente o resultado final, para não encontrar com problemas ou dificuldades muito grandes no meio do caminho.

Referências

ANTONIO, Luis. O Básico do C++. Disponível em: https://www.portalgsti.com.br/2013/04/apostila-de-c-disponivel-para-download-gratis.html. Acessado em: 10 de fevereiro de 2020.

CURRY, Caleb. C++ Programming. Disponivel em: https://www.youtube.com/watch?v=_bYFu9mBnr4&t=9s&ab_channel=CalebCurry. Acessado em: 15 de fevereiro de 2020.