Manual Instalação e Uso da Canvas2D

Cesar Tadeu Pozzer

26/10/2020

**Uso da Canvas2D**

A Canvas2D nada mais é que uma **API didática** (não é focada em desempenho) feita em cima das API OpenGL/Glut (figura abaixo) visando facilitar o desenvolvimento de aplicativos gráficos básicos. A Glut é uma API destinada à criação da janela da aplicação e faz o processamento dos eventos de mouse/teclado/renderização. A API OpenGL não oferece comandos para essas atividades que a Glut desempenha. Além da Glut, outras APIs desempenham o mesmo papel, como por exemplo, a GLFW (com demo disponível no material da disciplina).

Aplicação do usuário

Canvas2D

Glut

OpenGL

O papel da API Canvas2D é simplesmente tornar mais transparente o desenho de primitivas básicas, sem que seja necessário conhecimento do funcionamento das APIs Glut e OpenGL. Ela disponibiliza comandos para desenho de linhas, quadrados e pontos. Também permite a seleção de cores e impressão de textos. Pode ser usada para programar jogos 2D como Space invaders, Tetris, e muitos outros. Pode também ser usada para criação de aplicativos 3D e manipulação de imagens, porém com muito baixa performance.

Por ser baseada no OpenGL, toda programação é baseada em eventos.

|  |
| --- |
| Basicamente, na Canvas2D é feito um bypass dos comandos de teclado e mouse, implementados por meio de callbacks (na Glut isso é feito assim, na GLFW é diferente). Esses comandos são agrupados em apenas duas funções, com mais parâmetros. A canvas2D também faz a inicialização do OpenGL, definição da forma de renderização, que nesse caso é em 2D, e especificação de primitivas gráficas simples usando a sintaxe do OpenGL. |

A Canvas2D é fundamentada em três funções:

1. mouse() - Trata todos os eventos de mouse. A função é chamada quando ocorrer algum evento de mouse, seja movimento, arrasto, clique ou scrool.
2. keyboard() - Trata todos os eventos de keyboard. A função é chamada quando ocorrer algum evento de teclado.
3. render() – Único local onde devem ser colocados comandos para desenho de primitivas gráficas. Essa função é chamada continuamente. Dependendo da placa de vídeo e das configurações de v-sync, isso pode ocorrer mais de 1000 vezes por segundo.

A forma geral de uso da API é por meio de variáveis globais. Como exemplo, considere a implementação de uma entidade que se move controlada pelo teclado (como no jogo Pac-man). Na função keyboard() devem ser tratados os eventos das setas direcionais que movem a entidade na vertical e horizontal. O seguinte pseudo-código ilustra como deve ser tratado o evento para mover a entidade para a direita.

//variável global que controla a movimentação para a direita

bool g\_direita = false;

void render()

{

if (g\_direita == true)

Entidade.x++; //move a entidade para a direita

desenhaEntidade();

}

void keyboard(int tecla)

{

if (tecla == seta\_direita)

g\_direita = true; //muda o estado da variável global

else

g\_direita = false; //para de andar para a direita

}

Deve-se atentar para não colocar laços de repetição dentro da função render(), como no seguinte exemplo. Isso iria fazer a aplicação travar.

void render()

{

while (tecla == seta\_direita)

Entidade.x++;

}

Pode-se usar laços para fazer o desenho de várias entidades, como no seguinte exemplo.

void render()

{

for (linhas)

for (colunas)

desenhaEntidade(linha, coluna);

}

Para fazer o gerenciamento do que desenhar na canvas2D pode-se adicionar testes condicionais, como no seguinte exemplo, também fazendo uso de variáveis globais.

void render()

{

if (menu == true)

desenhaMenu();

if (morreu == true)

desenhaTelaFimJogo();

if (jogando == true)

desenhaCenarioJogo();

}

Todas os métodos da Canvas2D são estáticos. Seguem exemplos de uso:

CV::color(1);

CV::translate(100, 100);

CV::point(10, 10); //desenha na posição 110, 110.

CV::rectFill(Vector2(0, 0), Vector2(30, 30));

CV::text(10, 300, “Ola mundo”);

A função translate() simplifica o cálculo de coordenadas para desenho de muitos elementos relacionados entre si. Observe que o este comando não é cumulativo, ou seja, chamar translate(10,10), seguido de translate(20,20), não move a origem para coordenada (30,30), mas sim para a coordenada (20,20). Tudo o que vier a ser desenhado após o translate (funções point(), circle(), rect(), etc) sofrerá uma translação de (20,20).

Os seguintes exemplos ilustram o uso do comando CV::translate().

|  |  |
| --- | --- |
| Com | Sem |
| CV::translate(200, 200);  for(ang = 0; ang < 2\*PI; ang += 0.01)  {  x = cos(ang) \* 50;  y = sin(ang) \* 50;  CV::point(x, y);  } | for(ang = 0; ang < 2\*PI; ang += 0.01)  {  x = cos(ang) \* 50;  y = sin(ang) \* 50;  CV::point(200 + x, 200 + y);  } |

A inicialização da Canvas2D é bem simples:

int screenWidth = 500, screenHeight = 500; //largura e altura inicial da canvas2D

int main(void)

{

CV::init(&screenWidth, &screenHeight, "Titulo da Janela");

CV::run();

}

**Configuração da Canvas2D no Code::Blocks**

Para compilar/executar um aplicativo em OpenGL é necessário a existência de arquivos h, lib e dll. Todos os arquivos necessários para rodar em Windows já estão inseridos nos demos e configurados no arquivo de projeto.

<Linker>

<Add library="../lib/libglu32.a" />

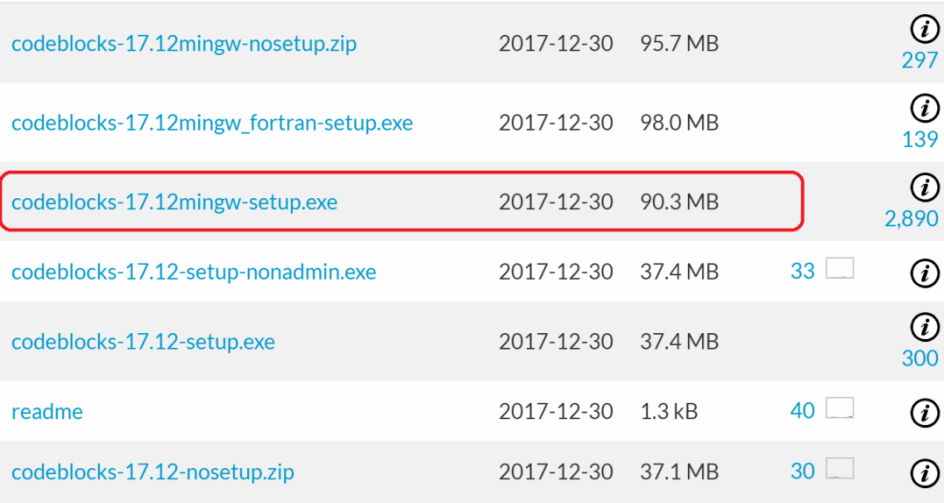
<Add library="../lib/libopengl32.a" />

<Add library="../lib/freeglut.lib" />

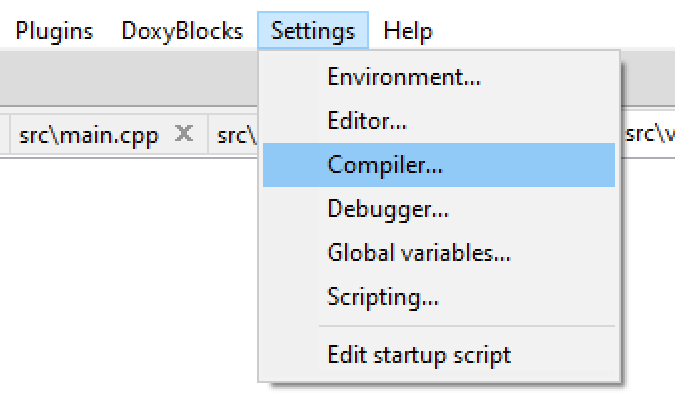
</Linker>

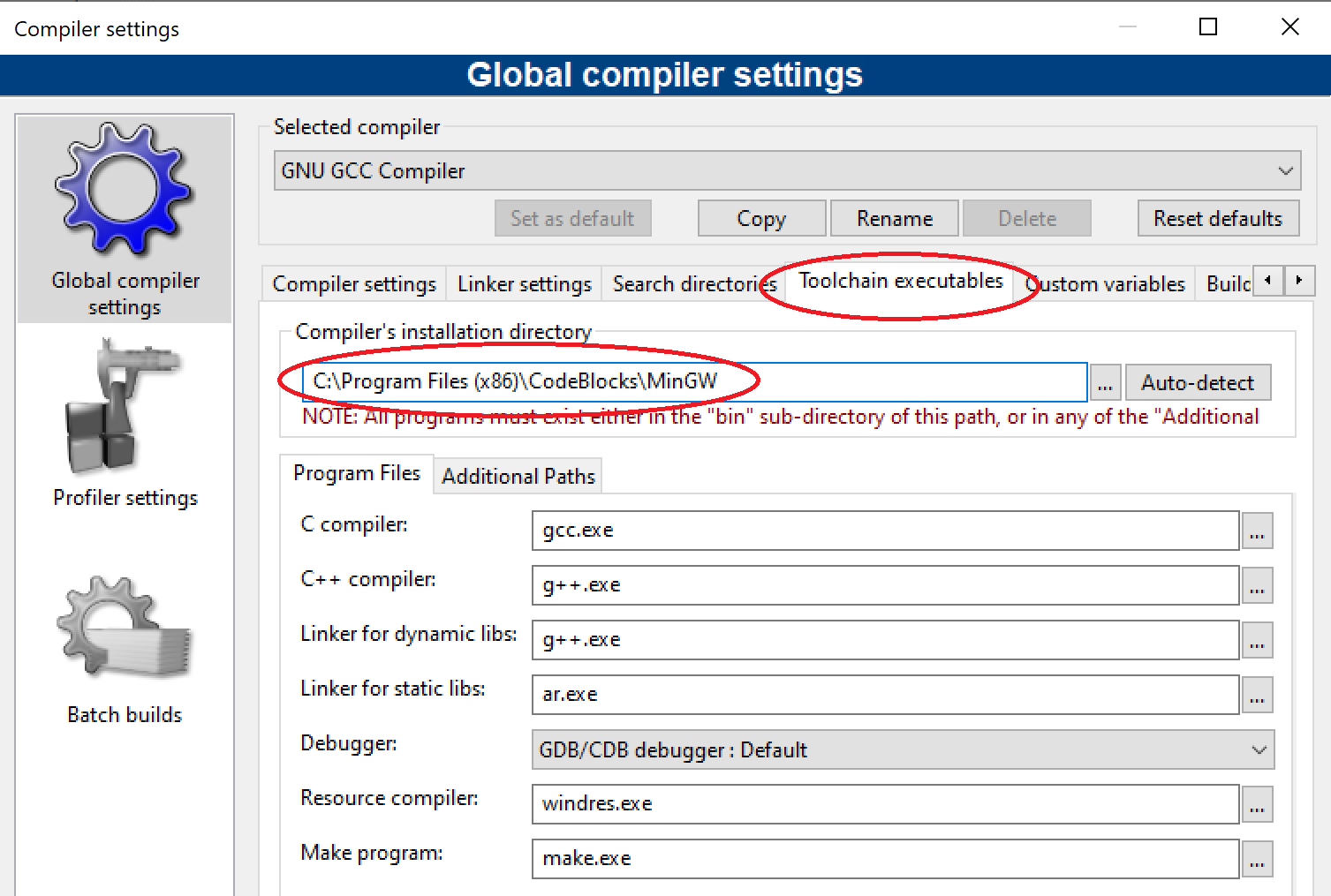
Por questões de compatibilidade de bibliotecas e dlls, recomenda-se o download da versão 17.12 do Code::Blocks, juntamente com o compilador MinGW. Deve-se baixar o arquivo codeblocks-17.12mingw-setup.exe.





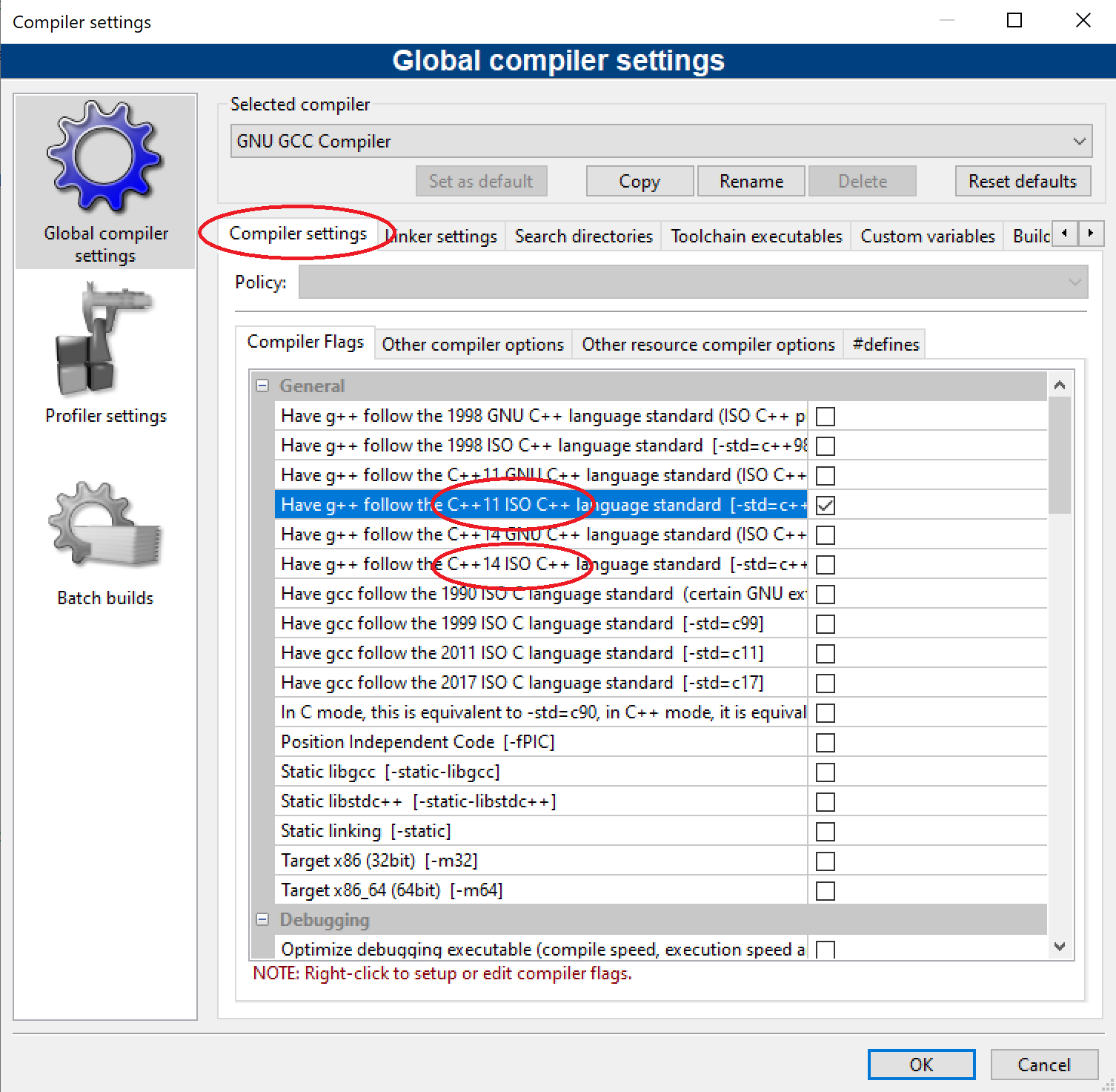
Após a instalação, deve-se certificar que o Code::Blocks esteja utilizando a versão do compilador MinGW que veio junto com o instalador.





Algumas vezes o Code::Blocks tem problemas na hora de fazer uma build por causa de arquivos de builds antigas. Procurem usar o comando Rebuild  do Code::Blocks quando isso acontecer.

Alguns programas podem requerer versões mais atuais da linguagem C++. Para ajustar:



Pode-se também alterar diretamente no arquivo de configuração do projeto:

<Target title="Release">

<Option output="../\_\_bin/Release/canvas" prefix\_auto="1" extension\_auto="1" />

<Option working\_dir="../" />

<Option object\_output="../\_\_obj/Release/" />

<Option type="1" />

<Option compiler="gcc" />

<Compiler>

<Add option="-std=c++11" />

<Add option="-O2 -Wall" />

<Add directory="include" />

</Compiler>

<Linker>

<Add option="-s" />

</Linker>

</Target>

**Configuração para Linux (Por Bruno Torres)**

Seguem algumas dicas para as distribuições do Linux derivadas do Debian ou Ubuntu.

--- Instalação de pacotes

Primeiramente, é preciso ter instalado o gcc/g++, OpenGL e FreeGLUT:

sudo apt-get install build-essential mesa-utils freeglut3-dev

Instalação do Code::Blocks:

sudo apt-get install codeblocks

Caso o pacote não seja encontrado, dê uma olhada neste link.

--- Alterações nos projetos do Code::Blocks

É preciso alterar algumas linhas nos projetos do CB para compilar no Linux. É só encontrar os arquivos com final .cbp e editar em qualquer editor de texto.

Você pode substituir as linhas ou apenas comentá-las (recomendado) usando <!-- e -->.

Original (Windows):

<Linker>

<Add library="..\lib\libglu32.a" />

<Add library="..\lib\libopengl32.a" />

<Add library="..\lib\freeglut.lib" />

</Linker>

No Linux, substituir por:

<Linker>

<Add option="-lGL" />

<Add option="-lGLU" />

<Add option="-lglut" />

</Linker>

Alguns projetos (e.g. demo gl\_14\_texture) também requerem que as linhas

<Add directory="include" />

dentro das tags <Compiler> sejam removidas ou comentadas. Caso contrário, o compilador irá tentar utilizar os headers que vem com os demos, que são para Windows.