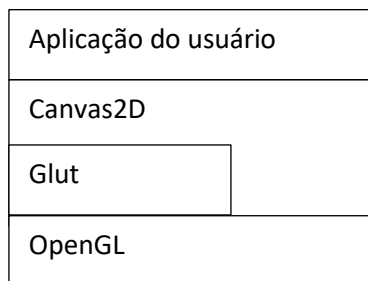


Manual Uso da Canvas2D

Cesar Tadeu Pozzer

2021

A Canvas2D nada mais é que uma API feita em cima das API OpenGL/Glut visando facilitar o desenvolvimento de aplicativos gráficos básicos (figura abaixo). 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).



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. 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 scroll.
- 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();
}
```

Seguem exemplos de uso:

```
color(1);
translate(100, 100);
point(10, 10); //desenha na posição 110, 110.
rectFill(Vector2(0, 0), Vector2(30, 30));
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 `translate()`.

Com	Sem
<pre>translate(200, 200); for(ang = 0; ang < 2*PI; ang += 0.01) { x = cos(ang) * 50; y = sin(ang) * 50; point(x, y); }</pre>	<pre>for(ang = 0; ang < 2*PI; ang += 0.01) { x = cos(ang) * 50; y = sin(ang) * 50; point(200 + x, 200 + y); }</pre>

A inicialização da Canvas2D é bem simples:

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

int main(void)
{
    init(&screenWidth, &screenHeight, "Titulo da Janela");
    run();
}
```

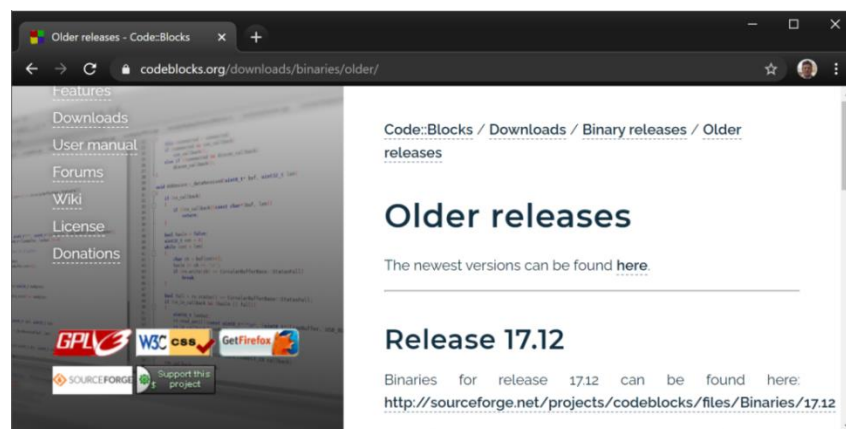
Configuração da Canvas2D no Code::Blocks

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 e configurados no arquivo de projeto (**canvas.cbp**). Esse arquivo pode ser visualizado em qualquer editor de texto.

```
<Linker>
  <Add library="../lib/libglu32.a" />
  <Add library="../lib/libopengl32.a" />
  <Add library="../lib/freeglut.lib" />
</Linker>
```

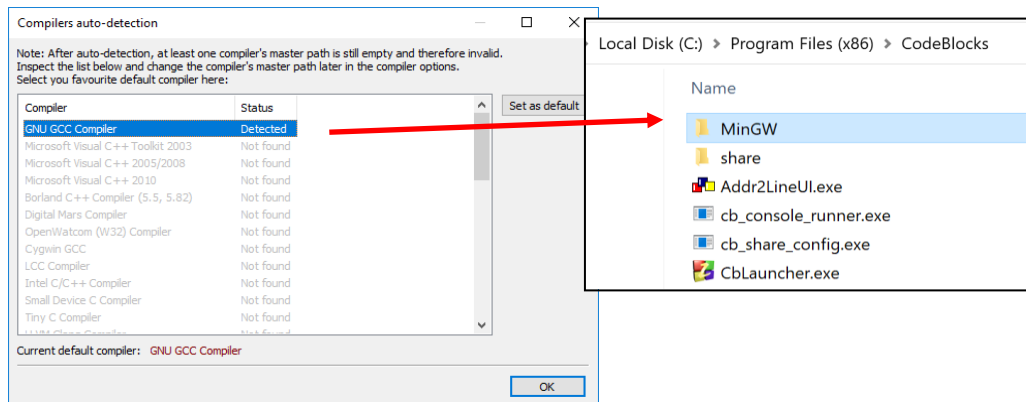
Por questões de compatibilidade de bibliotecas e dlls, deve-se fazer o download da versão **17.12** (88 MB) do Code::Blocks, juntamente com o compilador MinGW. Deve-se baixar o arquivo codeblocks-17.12mingw-setup.exe, no seguinte endereço:

<https://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/codeblocks-17.12mingw-setup.exe/download>

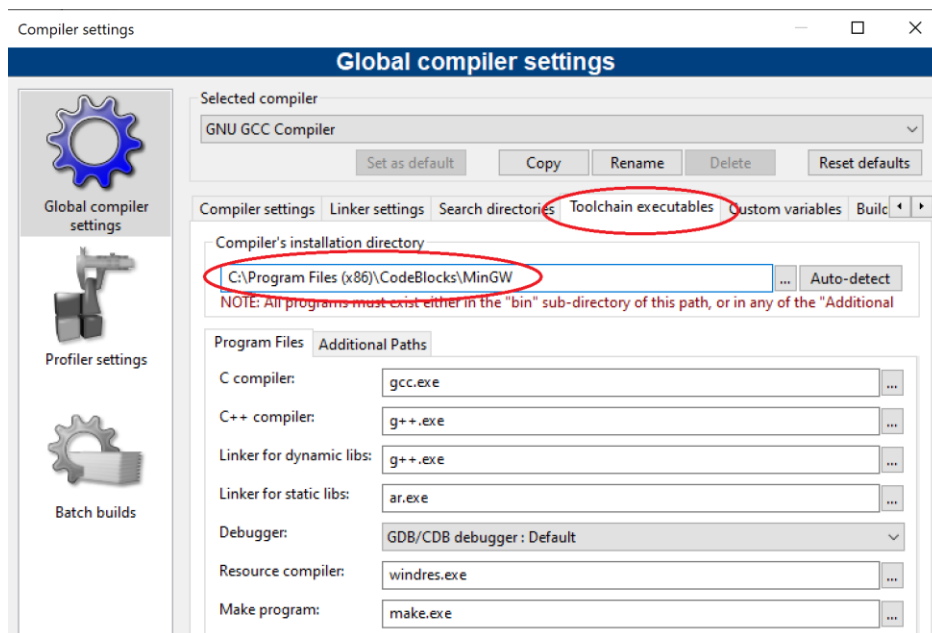
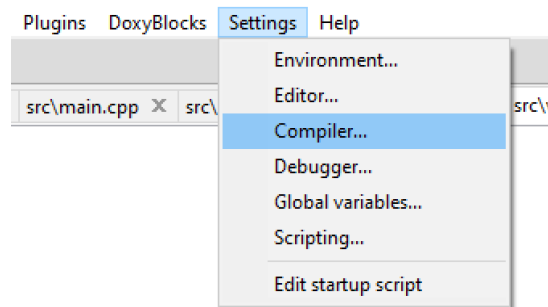



codeblocks-17.12mingw-nosetup.zip	2017-12-30	95.7 MB	297	i
codeblocks-17.12mingw_fortran-setup.exe	2017-12-30	98.0 MB	139	i
codeblocks-17.12mingw-setup.exe	2017-12-30	90.3 MB	2,890	i
codeblocks-17.12-setup-nonadmin.exe	2017-12-30	37.4 MB	33 <input type="checkbox"/>	i
codeblocks-17.12-setup.exe	2017-12-30	37.4 MB	300	i
readme	2017-12-30	1.3 kB	40 <input type="checkbox"/>	i
codeblocks-17.12-nosetup.zip	2017-12-30	37.1 MB	30 <input type="checkbox"/>	i

Ao instalar o Code::Blocks, tenha certeza que ele está utilizando o compilador MinGW, conforme ilustrado abaixo.

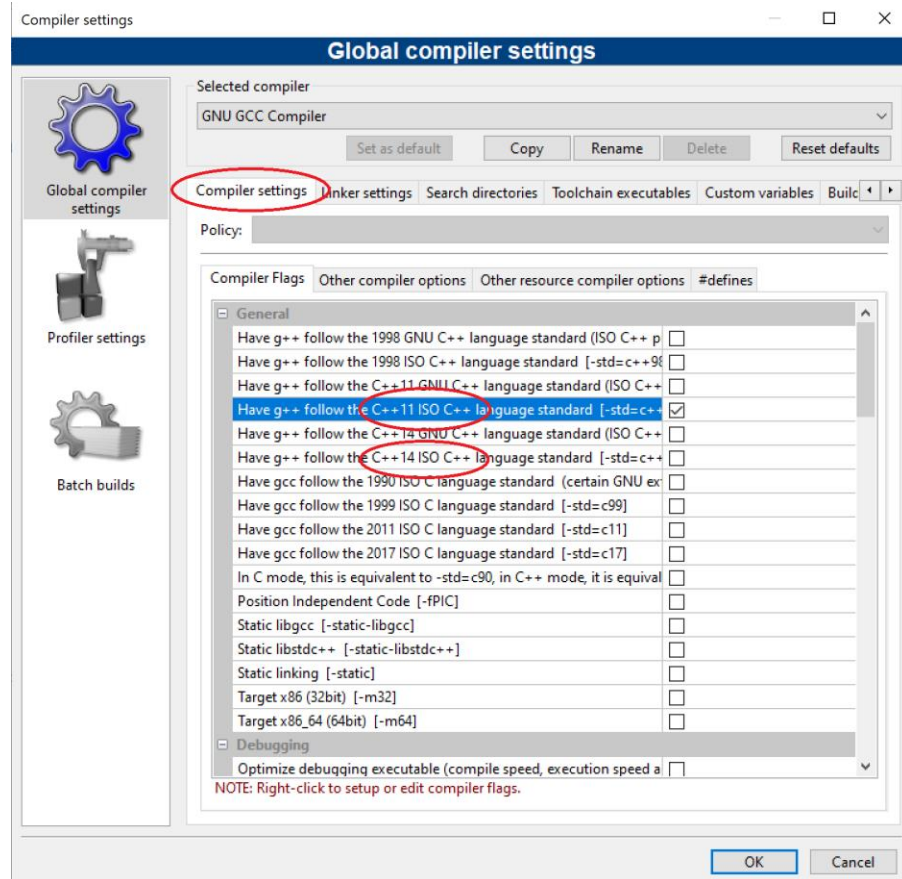


Após a instalação, deve-se certificar que o Code::Blocks esteja utilizando a versão do compilador MinGW que veio junto com o instalador.



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

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
```

--- 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>
```