Computação Gráfica

Aulas teóricas e práticas

Requisitos

Para a escola

Requisitos para as salas das aulas práticas de Computação Gráfica

- Windows XP Service Pack 2 ou Vista, ou
- (K)Ubuntu Linux 7.04 32bit e 64 bit, ou
- Mac OS 10.4 ou posterior.
- Instalação automática do Qt SDK para a plataforma escolhida, em `http://qt.nokia.com/downloads`. Alguns detalhes sobre a instalação em Windows:
    - Durante a instalação deixe que o Qt SDK instale também o MinGW, mesmo que já o tenha no seu computador. Ele não irá alterar o `PATH` e não irá interferir com a versão do MinGW já instalada.
    - Não é necessário o Qt SDK ser instalado com permissões administrativas nem estar na pasta `C:\Programas` ou `C:\Program Files`.
    - Após instalação, a configuração inicial do Qt Creator costuma estar incorrecta. Arranque o Qt Creator. Abra Tools / Options / Qt4 / Qt Versions. Carregue no botão "+" e escreva:
        - Version name: 4.6.2
        - QMake Location: instalação\qt\bin\qmake.exe
        - MinGW Directory: instalação\mingw

      onde instalação é o directório onde instalou o Qt SDK (se não especificou, foi `C:\qt`), e carregue em Apply. Depois escolha esta (4.6.2) como sendo a "Default Qt Version" e carregue em OK. Reinicie o Qt Creator antes de trabalhar com ele.
- Acesso à Internet com um *browser*.

Deve haver 1 PC por aluno.

Cada aula está programada para uma duração de 4h.

Para o aluno

Comparência nas aulas. Este guião tem propositadamente omissos certos elementos importantes para a compreensão total da matéria (notas históricas, relações entre partes diferentes da matéria, avisos sobre erros comuns, etc., ou seja, elementos para uma nota 20), embora seja suficiente para passar com nota bastante acima de 10.

Deves ter instalado o Qt SDK em computador próprio se quiseres acompanhar a matéria em casa. Vê a secção acima para requisitos e a primeira aula para instruções de instalação. Não é no entanto de todo necessário que tenhas estes sistemas em casa para conseguires passar à cadeira (podes usá-los na escola).

Esta cadeira assume que já tens experiência no uso de computadores (não necessariamente no Qt SDK) e em programação C ou C++.

Aula 01

Introdução e contextualização:

- O Qt e o seu SDK
- Revisões de Matemática.

Hiperligações.

Porquê Qt

As aulas práticas de Computação Gráfica vão ser sobre programação com OpenGL em ambiente gráfico de sistemas operativos modernos com a *framework* Qt (pronuncia-se "*cute*"). Existem duas razões para isto:

1. OpenGL é uma das principais normas para desenho de gráficos tridimensionais (e também, em menor importância, 2D), juntamente com o Direct3D da Microsoft. Mas ao contrário do segundo, OpenGL é uma norma aberta suportada universalmente. Podes ver mais informações (razoavelmente isentas) sobre a comparação entre ambos na Wikipedia (em Inglês): http://en.wikipedia.org/wiki/Comparison_of_OpenGL_and_Direct3D
2. Como não é o objectivo desta cadeira explicar-te programação em ambiente de janelas, e muito menos num sistema operativo específico de um só fabricante, é usada nas aulas práticas a *framework* Qt que permite que o mesmo programa seja compilado para as três principais plataformas. Assim, como mais valia das aulas, terás alguma experiência a criar aplicações Windows, Mac e Linux em ambiente de janelas.

Mais sobre o Qt (em Inglês) em http://qt.nokia.com/.

Os exemplos: CGMaze

CGMaze é uma aplicação desenvolvida especificamente para apoio a estas aulas. Trata-se de um "jogo" de labirinto que permite ao jogador "entrar" no labirinto e "passear" por ele com vista na 1ª pessoa.

O código-fonte do jogo é-te dado, com excepção de alguns ficheiros que terás de ser tu a completar e/ou acompanhar nas aulas. Estes ficheiros foram preparados para esconderem os detalhes Qt do código OpenGL em si, de forma a poderes concentrar-te no que mais interessa à cadeira.

Os exemplos: NeHe

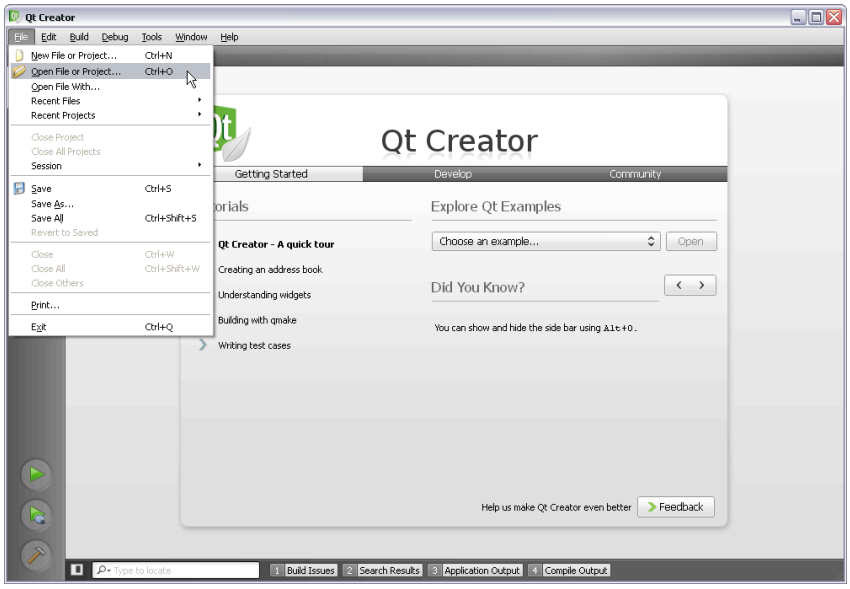
Vamos também estar a usar alguns tutoriais de OpenGL do *site* Neon Helium Productions (NeHe). Podes consultá-los em: http://nehe.gamedev.net/

Estes exemplos estão preparados para Windows (e Visual Studio), mas foram convertidos para a *framework* Qt no capítulo 14 do *Independent Qt Tutorial* em http://www.digitalfanatics.org/projects/qt_tutorial/.

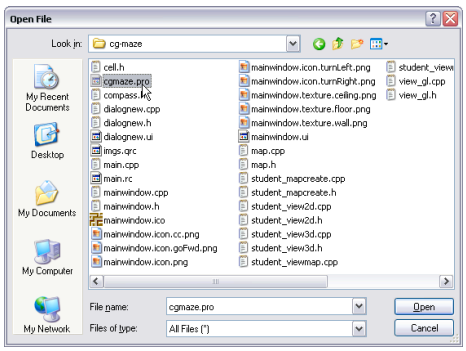
O código está disponível para ser descarregado no *site* do professor, já com as correcções para a última versão do Qt que estamos a usar.

Compilar um projecto Qt

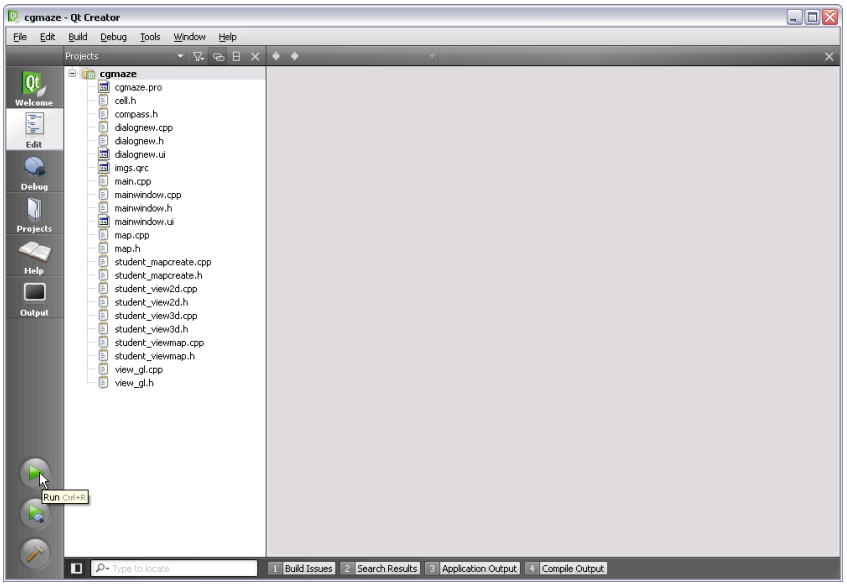
Arranca o Qt Creator que vais encontrar no teu menu Início / Programas. Escolhe do menu File, "Open File or Project":



Navega até ao directório onde descomprimiste o código-fonte do CGMaze e escolhe o ficheiro "cgmaze.pro". Esse é o ficheiro que descreve o projecto CGMaze.



Agora, com o projecto aberto, carrega no botão "Play" do fundo do ecrã para o projecto ser compilado *(built)* e correr.

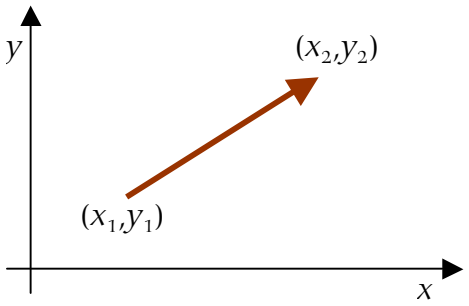


Na versão de código-fonte a que tens acesso, é normal que o jogo não funcione bem: não desenha o mapa em nenhuma das 3 vistas. Isso ficará para resolvermos nas aulas e tu resolveres em projecto.

Usas as mesmas técnicas para todos os projectos NeHe.

Revisões de Matemática: Vectores

Vectores são uma semi-recta entre dois pontos. Podem ser usados com uma direcção implícita, sendo então desenhados como uma seta do primeiro ponto para o segundo:



O vector é a linha exemplificada acima, ou seja, é a diferença entre os dois pontos.

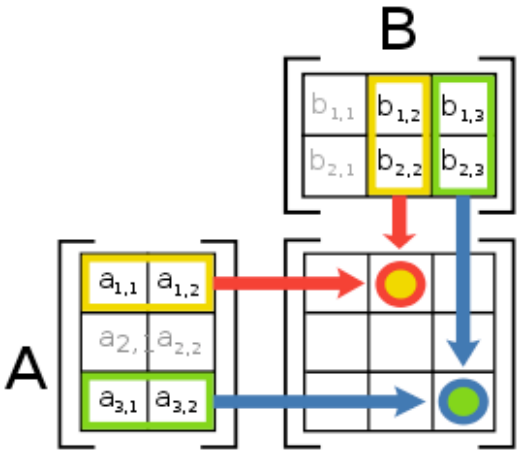
Como é uma diferença, a soma de um ponto com um vector dá outro ponto. Por exemplo, se somarmos ao ponto (x_1,y_1) acima o vector descrito no mesmo diagrama, o resultado é precisamente o ponto (x_2,y_2).

Da mesma forma posso ir somando vectores uns aos outros, o que é o mesmo que os ir desenhando em "comboio", a cauda do seguinte com cabeça do anterior. O resultado é o ponto na cabeça do vector final. O vector resultante dessa soma é

o que liga o primeiro ponto inicial ao ponto do resultado final (diagramas na aula).

Revisões de Matemática: Produto de Matrizes

Se eu tiver uma matriz **A** a multiplicar por uma matriz **B**, a matriz resultante tem tantas linhas quantas há em **A** e tantas colunas quantas há em **B**, e é dada por:



O valor na célula a amarelo é $a_{1,1}b_{1,2}+a_{1,2}b_{2,2}$ e o valor na célula verde é $a_{3,1}b_{1,3}+a_{3,2}b_{2,3}$.

O número de colunas em **A** tem de ser igual ao número de linhas em **B** para a multiplicação poder existir.

A multiplicação de matrizes é associativa, ou seja

$$(\mathbf{M}_1 \times \mathbf{M}_2) \times \mathbf{M}_3 = \mathbf{M}_1 \times (\mathbf{M}_2 \times \mathbf{M}_3)$$

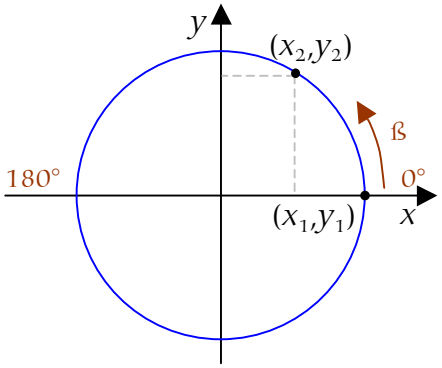
No entanto a multiplicação de matrizes não é habitualmente comutativa, ou seja, geralmente

$$\mathbf{M}_1 \times \mathbf{M}_2 \neq \mathbf{M}_2 \times \mathbf{M}_1$$

Podes ler muitos mais detalhes sobre a multiplicação de matrizes na Wikipedia em: http://pt.wikipedia.org/wiki/Produto_de_matrizes. A versão em Inglês tem muita informação.

Revisões de Matemática: Trigonometria

Trigonometria descreve operações que, com base num ângulo de rotação, nos dão medições da nova posição (x_2,y_2) de um ponto (x_1,y_1), depois de ser rodado.



O círculo trigonométrico assume que as rotações são feitas à volta do eixo de coordenadas (i.e., (0,0)), e que o raio de rotação mede 1. Assim sendo, (x_1,y_1) = (1,0) e por definição x_2=cos(ß) e y_2=sin(ß)*.

Os ângulos são medidos em graus (em OpenGL): 0° define a posição indicada por (x_1,y_1) (Leste na Rosa-dos-Ventos), 90° é ¼ de rotação no sentido inverso ao dos ponteiros do relógio, ou seja Norte, 180° é meia rotação (Oeste) e assim por diante. 360° é uma rotação completa e válido, mas igual a 0°. Também podemos fazer os ângulos na direcção oposta e por exemplo -90° = 270°.

Exercícios

Qual é o resultado e aspecto visual de:

- Vector resultante da soma dos vectores (-1,1) com (1,1)
- Posição final do ponto (2,3) após lhe serem somados os vectores descritos acima

Qual é a matriz resultante do produto de:

- $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$
- $\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

* Ou em Português, sen(ß), mas como a biblioteca matemática do C (como na maior parte das linguagens de programação) está em Inglês, a função chama-se sin() (do Inglês *sine*).

- $\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$

Qual é o valor de:

- sin(0°)
- cos(90°)
- sin(270°)
- sin(45°)² + cos(45°)²
- cos(180°)

Olha para o código do CG Maze.

- Qual é o tipo de dados (classe) que guarda a informação de cada célula do mapa? Se eu tiver uma variável `c` que descreve uma dessas células, como é que eu:
  - Altero `c` para indicar que o jogador 1 passou por lá
  - Se `c` descreve uma porta fechada, como é que o altero para passar a descrever uma porta aberta?
  - Se eu atribuir o número 3 a um objecto do jogo (e.g.: um machado), como é que eu coloco esse objecto em `c`?
- Qual é o tipo de dados (classe) que guarda a orientação que o jogador tem no mapa? Se eu tiver uma variável `c` que descreve uma tal orientação, como é que eu:
  - Altero `c` para indicar que o jogador rodou para a direita
  - Altero `c` para indicar que o jogador rodou para trás
  - Converto `c` num número real (`double`) que tem essa orientação mas em graus medidos no círculo trigonométrico
- Qual é o tipo de dados (classe) que guarda a grelha de jogo (o conjunto de células pertencentes ao mapa)? Se eu tiver uma variável `m` que descreve uma grelha, como é que eu:
  - Leio a célula na posição genérica (`x`,`y`)
  - Altero a célula na posição genérica (`x`,`y`)
  - Verifico se a posição (`x`,`y`) está dentro dos limites da grelha
  - Obtenho a largura da grelha
  - Verifico se é possível para o jogador actual mover-se para a posição (`x`,`y`)
  - Movo o jogador actual que está na posição (`x`,`y`), para a frente

    - Movo o jogador actual que está na posição (`x`,`y`), lateralmente para a esquerda
- A janela principal do jogo é desenhada visualmente no ficheiro `mainwindow.ui`. Abre-o e localiza depois em que ficheiros estão as classes que são responsáveis por desenhar as vistas:
    - Do mapa
    - Em 1ª pessoa, usando OpenGL 2D
    - Em 1ª pessoa, usando OpenGL 3D
- A partir do ponto anterior, descobre onde é que deves escrever código para:
    - Fazer a inicialização OpenGL da vista em 1ª pessoa com OpenGL 2D
    - Desenhar uma célula da grelha na vista do mapa nas coordenadas (`x`,`y`)
    - Desenhar a vista interior do labirinto a partir da posição (`x`,`y`) e orientação específica (em 2D e 3D)

Bom trabalho!

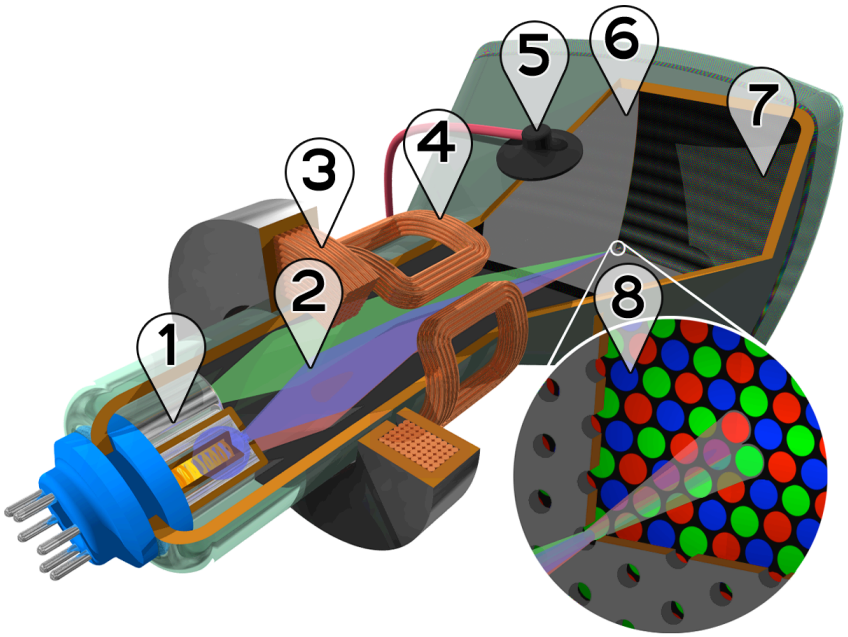
Aula 02

Hardware: Tipos de ecrãs. CRT, LCD, LED. HD e Interlacing. Matriz de *pixels*, cores *(sub-pixels)*, varrimento, fotogramas. VGA, DVI, HDMI. Ecrãs estereoscópicos e 3D.

Ilusão de animação. Fotogramas.

Ecrãs CRT

Tubos de raios catódicos *(Cathode-Ray Tube)*. Ilustração[*]:



Onde:

1. Emissor de electrões.
2. Feixes de electrões.
3. Bobina de foco.
4. Bobinas de deflexão.
5. Conexão do ânodo (pólo positivo).
6. Tela para separação dos raios para o vermelho, verde e azul da imagem mostrada.

[*] Imagem retirada da Wikipedia; autor "Grm wnr".

7. Camada de fósforo com áreas vermelhas, verdes e azuis.
8. Detalhe do interior da tela revestida de fósforo.

Conceitos básicos OpenGL: Sistema de coordenadas

Em OpenGL desenhamos figuras num sistema de coordenadas ideal, com a origem no canto inferior esquerdo, tal como num gráfico em Matemática.

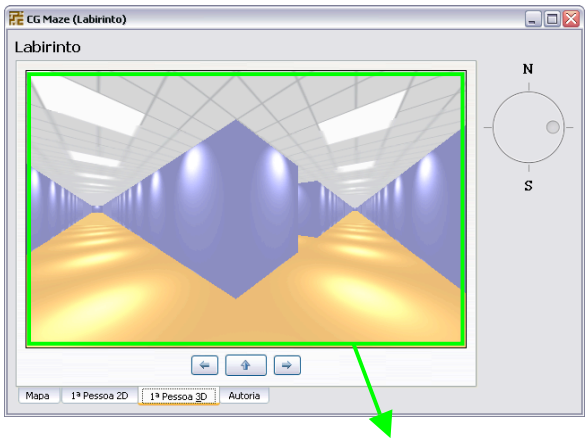
As coordenadas dos pontos são números reais (embora também possa usar números inteiros), tão precisos quanto o programador precise (dentro da capacidade de um `float` ou `double`). Caberá ao OpenGL fazer a conversão deste "mundo ideal" para os *pixels* do ecrã através de projecções (explicadas em aula posterior).

Janelas e *viewports*

Não vamos poder escrever em todo o ecrã do nosso computador. Para além de isso interferir com a funcionalidade de outras aplicações a correr ao mesmo tempo no ecrã, isso poderia ser até um problema de segurança.

Os sistemas operativos costumam separar visualmente cada aplicação em rectângulos independentes. A terminologia do sistema operativo para este rectângulo é de "janela".

Cada aplicação pode por sua vez separar a sua janela em várias secções rectangulares independentes (para desenhar botões, zonas de desenho, etc.). Cada vez que um método da aplicação tem uma dessas secções activa para lá escrever, chama a essa secção de *viewport*.



Viewport do OpenGL
dentro da janela do jogo.

Em OpenGL este *viewport* define-se com:

```
glViewport( x, y, width, height );
```

onde `x` e `y` são as coordenadas do canto inferior esquerdo do *viewport* OpenGL em relação ao *viewport* da aplicação onde o OpenGL pode desenhar `width` e `height` são a sua largura e altura, respectivamente. Todos estes valores devem ser inteiros já que se referem a coordenadas de *pixels* no ecrã.

Toda e qualquer tentativa de desenhar fora do *viewport* é gorada. Ao "corte" de linhas e outras figuras dentro de um *viewport* ou janela chama-se *clipping*.

Primitivas 2D

Tendo agora um sistema de coordenadas posso começar a desenhar figuras. Por exemplo, desenho um quadrado entre as coordenadas (0,0) e (1,1) com:

```
glBegin( GL_QUADS );
      glVertex2f( 0, 0 );  // Canto inferior esquerdo
      glVertex2f( 0, 1 );  // Canto superior esquerdo
      glVertex2f( 1, 1 );  // Canto superior direito
      glVertex2f( 1, 0 );  // Canto inferior direito
glEnd();
```

Ou seja, indico o tipo de figura que vou desenhar com `glBegin()`, indico a posição de cada vértice com `glVertex2f()` (ou suas variantes), e termino a sequência com `glEnd()`.

`glBegin()` aceita vários argumentos para vários tipos de primitivas 2D, como pontos:

- `GL_POINTS` – Trata cada vértice como um ponto independente a ser desenhado.

Linhas e curvas:

- `GL_LINES` – Trata cada par de vértices como um segmento de linha independente a ser desenhado.
- `GL_LINE_STRIP` – Desenha um grupo de linhas ligadas umas às outras. Os primeiros dois vértices definem o primeiro segmento de linha. O segundo vértice e o terceiro definem o segundo segmento de linha e assim por diante. Se escolher bem os pontos, este argumento permite desenhar arcos (semi-círculos).
- `GL_LINE_LOOP` – Semelhante ao anterior, mas é também desenhado um segmento de linha do último vértice até ao primeiro, fechando a linha *(loop)*. Se escolher bem os pontos, este argumento permite desenhar círculos e elipses.

E figuras preenchidas:

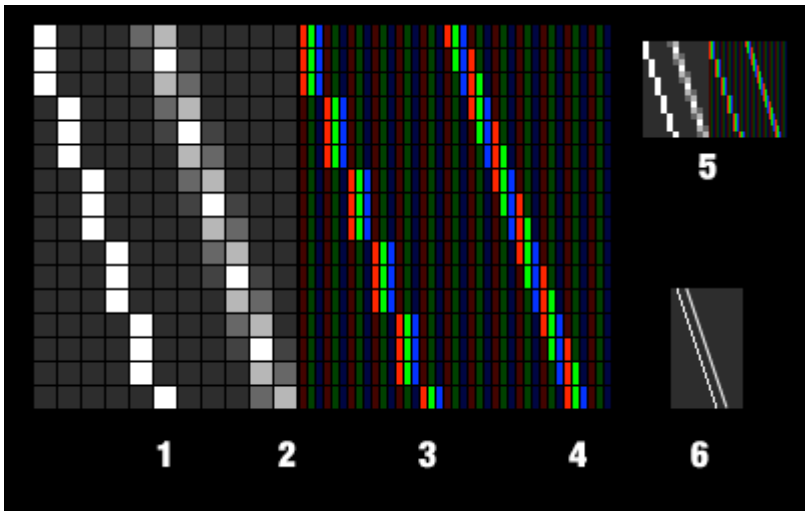
- `GL_TRIANGLES` – Trata cada conjunto de três vértices (i.e., cada "triplo" de vértices) como um triângulo independente.
- `GL_TRIANGLE_STRIP` – Desenha um grupo de triângulos colados. Depois dos primeiros três vértices, cada novo vértice faz um triângulo com os anteriores dois. Atenção à ordem dos vértices quando estiveres a usar texturas: consulta a documentação [OGL21].
- `GL_TRIANGLE_FAN` – Desenha um grupo de triângulos colados que partilham todos o primeiro vértice, daí o nome (leque, ou *fan* em Inglês). Se escolher bem os pontos, este argumento permite desenhar "fatias de bolo", círculos preenchidos (discos) e elipses preenchidas.
- `GL_QUADS` – Trata cada conjunto de quatro vértices (i.e., cada "quádruplo" de vértices) como um quadrilátero independente.
- `GL_QUAD_STRIP` – Desenha um grupo de quadriláteros colados. Depois dos primeiros quatro vértices, cada novo par de vértices faz um quadrilátero com os anteriores dois. Atenção à ordem dos vértices quando estiveres a usar texturas: consulta a documentação [OGL21]. Se escolher bem os pontos, este argumento permite desenhar "donuts" (toros ou aros).
- `GL_POLYGON` – Desenha um polígono convexo. Igual a `GL_LINE_LOOP`, mas preenche a figura.

Existem algumas funções OpenGL utilitárias (que começam com o prefixo "`glu`") que auxiliam na criação de curvas e discos, sem precisarmos de ter o CPU a calcular senos e co-senos. Não iremos dar essas funções, mas pesquisa por `gluBeginCurve()`, `glMap1f()` e funções relacionadas.

Anti-aliasing e ClearType

Como os ecrãs são construídos como uma matriz de pontos, linhas que não sejam absolutamente horizontais ou verticais não ficam com aspecto perfeito e têm de ser aproximadas. Se os pontos que compõem o ecrã não forem em quantidade suficiente (para serem de tamanho muito pequeno) esta aproximação é perceptível como defeito na qualidade visual da linha, já que esta terá um aspecto de "escada" *(aliasing)*.

Existem duas formas de resolver este problema[*]:



Na figura acima, 1 representa uma linha diagonal com o normal efeito de *aliasing*. Nós podemos no entanto, conforme a linha passa de uma coluna de *pixels* para a outra, "pintar" *pixels* adjacentes com tons de cinzento, conforme a linha "real" passa parcialmente em cada um. O resultado está ampliado em 2 e a este efeito de suavização chamamos de *anti-aliasing*.

No entanto, em figuras muito pequenas como texto com tamanho muito pequeno isto não chega e os vários *pixels* cinzentos fazem o texto parecer "desfocado" e difícil de ler. Uma solução é pensarmos em cada uma das três componentes RGB como "*sub-pixels*" independentes e desenhar a linha nesta matriz "aumentada". Na figura acima, 3 tem a linha original desenhada na matriz aumentada, 4 a linha desenhada recorrendo aos *sub-pixels*.

Texto desenhado com este método (chamado de ClearType em Windows e Quartz no Mac) não é universalmente aplaudido já que pessoas diferentes têm sensibilidade diferente às cores que ficam nas orlas das letras.

Activa-se o efeito *anti-aliasing* do OpenGL com

```
glEnable( GL_LINE_SMOOTH );
```

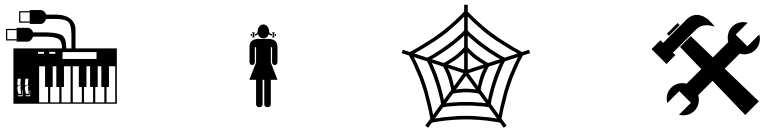
[*] Imagem retirada da Wikipedia, em http://en.wikipedia.org/wiki/ClearType.

O método ClearType/Quartz de usar *sub-pixels* não está disponível em nenhum sistema operativo para primitivas de desenho, apenas para texto. OpenGL não o suporta sequer em texto.

Exercício

Compila e corre o exercício NeHe ch2.

- Altera a largura e altura da janela onde surgem as figuras. O que acontece às figuras?
- A função OpenGL `glVertex3f()` desenha um vértice de uma figura.
    - Encontra o ficheiro onde ela se encontra
    - Faz um desenho em papel que mostra cada figura no seu próprio sistema de coordenadas
    - Altera o código para que o triângulo tenho o dobro da altura e o quadrado passe a ser um rectângulo que tem 4× a largura actual.
- O que aconteceria se eu substituísse a função `glViewport()` do método `resizeGL()` do ficheiro `ch2.h` por `glViewport(0,0,50,50)`? Testa a tua resposta fazendo essa substituição.
- No ficheiro `ch2.h`, apaga tudo da função `paintGL()` excepto as duas primeiras funções OpenGL (`glClear()` e `glLoadIdentity()`). Acrescenta a seguir uma linha `glTranslatef(0,0,-10)`. Assume que a origem do sistema de coordenadas está ao centro do ecrã e vai até 3.0 em cada direcção. Depois de `glLoadIdentity()` acrescenta instruções para desenharem da forma mais fiável possível:
    - Uma casa (um quadrado com um triângulo por cima)
    - Um losango (ou seja, um quadrado rodado 45º)
    - Uma estrela de 5 pontas (★) e uma estrela de David (✡)
    - A seguintes figuras da forma mais fiel possível:



Bom trabalho!

Aula 03

Translações, redimensionamentos, rotações; composição de transformações (matriz).

Transformações em sequência: Conceitos de animação *(double buffering*, fotogramas, fps, lapso entre fotogramas).

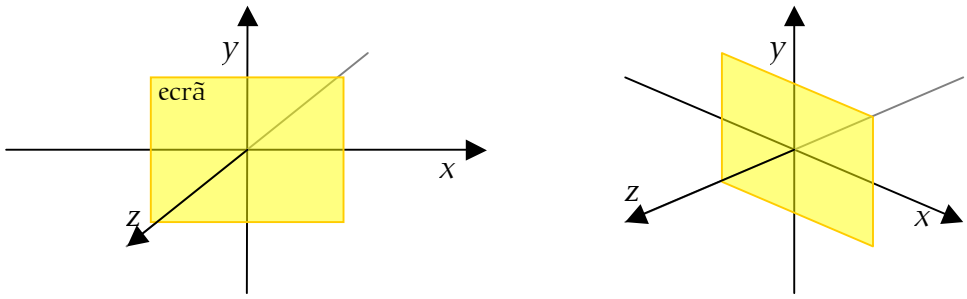
Lidar com temporizações no Qt. Exemplos no CGMaze.

3D

"3D" é a sigla de "3 Dimensões" e refere-se a objectos que têm volume. Para além das duas dimensões expressas numa folha de papel (*x* e *y*, ou largura e altura) temos também a profundidade (*z*), em 3D. O mundo à nossa volta pode ser todo descrito num sistema de coordenadas 3D.

Sistema de coordenadas 3D

Em OpenGL, o sistema de coordenadas tridimensional inclui as coordenadas *x* e *y* já nossas conhecidas, e *z*, profundidade, que é positiva para "fora" do ecrã, na direcção do observador e negativa para "dentro" do ecrã.



Quando trabalhamos com 3D, é habitual pensarmos na origem das coordenadas ao centro do ecrã. Embora possamos mudar isso de uma série de maneiras, verás que esta é a forma mais fácil de trabalhar com OpenGL inicialmente.

Primitivas 3D

Todas as primitivas 2D de OpenGL podem aceitar 3 coordenadas em vez de 2, transformando-as em primitivas 3D. É só trocar o "2" que faz parte do nome da função por "3" e acrescentar uma 3ª coordenada (*z*) ao final.

Matriz de transformação

Todas as transformações que eu posso aplicar a coordenadas/pontos 3D (translação, redimensionamento, rotação), podem ser feitas multiplicando uma matriz 4×4 com a transformação respectiva por um vector* com as coordenadas tridimensionais, mais uma coordenada, *w*, que terá inicialmente o valor 1.

Exemplo:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ w_2 \end{bmatrix}$$

Este exemplo mostra a matriz identidade que não faz qualquer alteração às coordenadas. Eu "carrego" essa matriz identidade para o sistema OpenGL com

```
glLoadIdentity();
```

Nota que estas matrizes aplicam a transformação **antes** de desenhar o respectivo ponto. Elas servem para eu transformar o objecto antes dele ser desenhado o ecrã, e não para transformar um objecto já desenhado no ecrã.

Translação de coordenadas

Se eu precisar de fazer uma translação a um objecto (mover o objecto) composto por vários pontos com coordenadas *x*, *y* e *z*, uso a seguinte matriz de transformação antes de definir os pontos:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cada um dos factores d_k é somado à coordenada respectiva.

Faço uma translação em OpenGL com a função

```
glTranslatef( dx, dy, dz );
```

onde `dx`, `dy` e `dz` são `floats`, ou

```
glTranslated( dx, dy, dz );
```

onde `dx`, `dy` e `dz` são `doubles`. Vê no entanto a secção abaixo sobre "composição de transformações".

* Também chamado de vector coluna por só ter uma coluna.

Redimensionamento de coordenadas

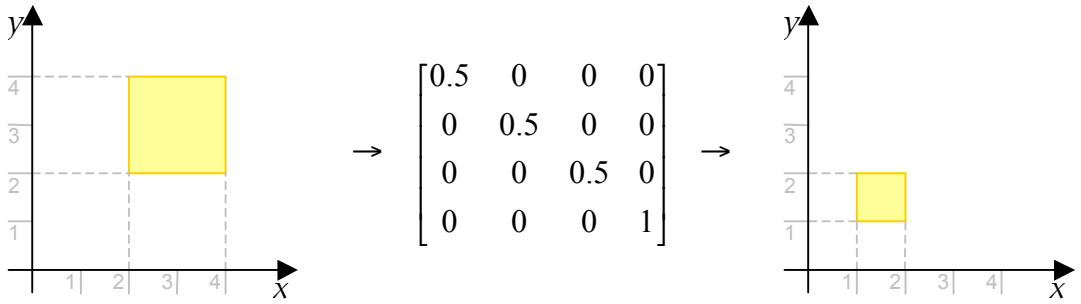
Se eu precisar de alterar as dimensões proporcionais (ou seja, ampliar ou reduzir) um objecto composto por vários pontos com coordenadas x, y e z, uso a seguinte matriz de transformação antes de definir os pontos:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cada um dos factores s_k é multiplicado pela coordenada respectiva. Se eu quiser redimensionar um objecto para passar a ter metade do tamanho, uso 0.5 para todos os valores de s_k. Se quiser triplicar a sua largura (x) e duplicar a altura e profundidade (y e z), uso s_x=3, s_y=2 e s_z=2.

Nota que este redimensionamento é relativo à origem de coordenadas. Se o objecto não estiver centrado na origem, ele também vai mudar de posição, aproximando-se ou afastando-se da origem.

Repara como este quadrado é redimensionado (reduzido) para metade do tamanho, mas também se "move" para mais perto da origem das coordenadas:



Resolvo este problema tendo os meus modelos dos objectos centrados na origem das coordenadas. Mais sobre isto, abaixo.

Faço um redimensionamento em OpenGL com a função

```
glScalef( sx, sy, sz );
```

onde `sx`, `sy` e `sz` são `floats`, ou

```
glScaled( sx, sy, sz );
```

onde `sx`, `sy` e `sz` são `doubles`. Vê no entanto a secção abaixo sobre "composição de transformações".

Rotação de coordenadas

Existem três matrizes diferentes para rotação de um objecto composto por vários pontos com coordenadas x, y e z, em torno de cada um dos eixos (x, y ou z). As

matrizes de transformação devem ser usadas antes de definir os pontos do objecto.

Para rodar no eixo dos *x*, uso a seguinte matriz:

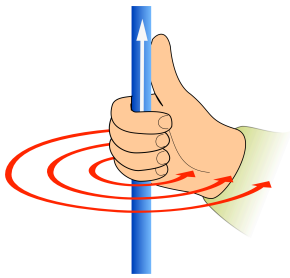
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_x & -\sin\alpha_x & 0 \\ 0 & \sin\alpha_x & \cos\alpha_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para rodar no eixo dos *y*, uso a seguinte matriz:

$$\begin{bmatrix} \cos\alpha_y & 0 & \sin\alpha_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha_y & 0 & \cos\alpha_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para rodar no eixo dos *z*, uso a seguinte matriz:

$$\begin{bmatrix} \cos\alpha_z & -\sin\alpha_z & 0 & 0 \\ \sin\alpha_z & \cos\alpha_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

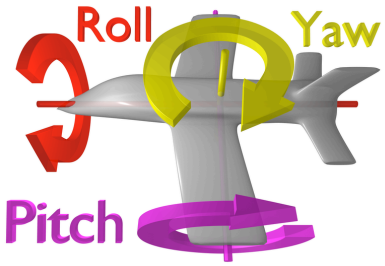


Cada rotação precisa de ter um argumento α_x, α_y ou α_z, que são os ângulos (em graus) de rotação. A direcção dos ângulos é determinada segundo a regra da mão direita[*]: coloco a mão direita como se estivesse a segurar num copo, com o polegar para cima.

Agora, sem mover os dedos da mão, alinho o polegar com cada um dos eixos. A direcção em que os outros 4 dedos apontam é a direcção de rotação de ângulos positivos, e vice-versa. Por exemplo, um α_z negativo faz um objecto rodar na direcção dos ponteiros do relógio.

[*] Imagem retirada da Wikipedia; autor "Jfmelero".

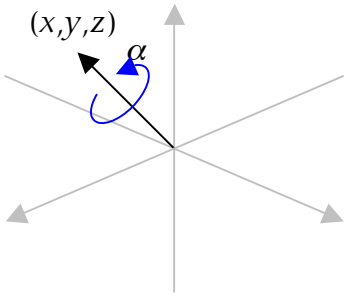
Se eu tiver um modelo de avião* "visto de baixo e a voar para a direita" (ou seja, paralelo ao plano de $z=0$ e a "voar" na direcção do eixo dos x), então ele está orientado nos eixos de inércia e em Inglês α_z é o *yaw* ou *heading* (guinada), α_x é o *roll* (rotação) e α_y é o *pitch* (também conhecido como *trim* em submarinos, ou arfagem em português).



Estas notações são frequentes quando usamos OpenGL em programação de jogos.

Para fazer uma rotação combinada em 2 ou mais eixos, a ordem em que são feitas as rotações é importante.

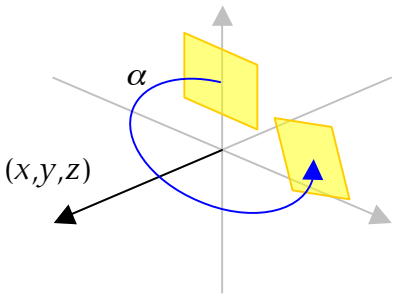
Também por esta razão, a rotação em OpenGL não se faz desta forma. Em vez disso definimos um vector (x, y, z) que será o eixo de rotação, assim como o ângulo (α) de rotação.



Assim, se o vector for (1,0,0) por exemplo, a rotação faz-se em torno do eixo dos x. A direcção dos ângulos continua a determinar-se pela regra da mão direita.

* Imagem retirada da Wikipedia; autor "ZeroOne".

Nota que todos estes eixos de rotação passam pela origem de coordenadas, e não necessariamente pelo objecto/ponto em si. Se o objecto não estiver centrado na origem, ele também vai mudar de posição, num círculo que passa pelo objecto e é perpendicular ao e centrado no eixo respectivo.



Resolvo este problema tendo os meus modelos dos objectos centrados na origem das coordenadas. Mais sobre isto, abaixo.

A operação faz-se com a função

```
glRotatef( angulo, x, y, z );
```

onde `angulo`, `x`, `y` e `z` são `floats`, ou

```
glRotated( angulo, x, y, z );
```

onde `angulo`, `x`, `y` e `z` são `doubles`. Vê no entanto a secção abaixo sobre "composição de transformações".

Composição de transformações

Um conceito importante em programação OpenGL é que existe sempre activa uma "matriz de transformação" que é aplicada a todas as coordenadas que são enviadas para o sistema.

Esta é uma única matriz que aglomera todas as transformações que foram pedidas pela aplicação. Isto porque, se chamarmos **M** a uma matriz de transformação e **C** ao vector de coordenadas, então quando aplicamos duas transformações estamos a fazer:

$$\mathbf{M}_2 \times (\mathbf{M}_1 \times \mathbf{C})$$

Como a multiplicação de matrizes é associativa,

$$\mathbf{M}_2 \times (\mathbf{M}_1 \times \mathbf{C}) = (\mathbf{M}_2 \times \mathbf{M}_1) \times \mathbf{C}$$

Eu posso então combinar ou compor todas as transformações que preciso de fazer numa só matriz e usar apenas essa com todos os vectores de coordenadas de que precisar!

Uso das transformações em OpenGL

As funções OpenGL que começam com `glLoad...` substituem a matriz de transformações. As restantes funções (`glTranslatef()`, etc.) aplicam a sua transformação a essa matriz, fazendo

$$\mathbf{M} = \mathbf{M} \times \mathbf{M}_{transf}$$

onde $\mathbf{M}_{transf}$ é a nova matriz da transformação a ser aplicada. Como a multiplicação de matrizes não é comutativa, esta operação equivale a aplicar primeiro a nova transformação $\mathbf{M}_{transf}$ e só depois a transformação $\mathbf{M}$ (vê como acima, aplicar o produto $\mathbf{M}_2 \times \mathbf{M}_1$ ao vector de coordenadas $\mathbf{C}$ equivale a aplicar primeiro a transformação $\mathbf{M}_1$ seguida da transformação $\mathbf{M}_2$).

Ou seja, **as transformações têm de ser especificadas pela ordem inversa no OpenGL!**

Por outro lado, isto significa que não é fácil "desligar" apenas algumas das transformações que temos activas. Por isso mesmo, o OpenGL dá-nos funções que guardam a matriz de transformações actual numa pilha (até 32 posições) de onde podem ser recuperadas.

As funções são:

```
glPushMatrix();
glPopMatrix();
```

Isto é preferível a tentar aplicar a transformação inversa porque é mais rápido e evita erros de arredondamento.

Outro cuidado a ter é ter os modelos dos objectos guardados na aplicação com coordenadas baseadas na origem (de forma a ser fácil redimensioná-los e rodá-los). De cada vez que temos de desenhar uma cena OpenGL, aplicamos as transformações adequadas para o colocar na posição final da cena.

Isto costuma fazer-se pela seguinte ordem:

```
glTranslatef( dx, dy, dz );
glRotatef( angulo, x, y, z );
glScalef( sx, sy, sz );
```

Esta técnica também é útil quando precisamos de ter 2 ou mais objectos idênticos em posições diferentes da nossa cena.

Transformações para colocação de câmara na cena

Em muitas situações queremos que a "câmara" que "filma" a cena 3D esteja posicionada dentro dessa mesma cena. OpenGL não suporta "câmaras" (ou melhor, a câmara está sempre na origem das coordenadas, a olhar na direcção inversa à do eixo dos *z*), pelo que isto faz-se movendo e rodando toda a cena de forma a posicionar os objectos no sitio correcto.

Por exemplo, se o jogador olha para a direita, então podemos (antes de desenhar qualquer objecto OpenGL) aplicar uma rotação dos objectos 90% para a esquerda.

Assim, apesar da câmara estar no mesmo sítio (origem das coordenadas), vemos à nossa frente os objectos que estão à direita.

Existe uma função auxiliar que nos ajuda a "definir a posição da câmara", criando na realidade as rotações e translações adequadas para "fingir" essa colocação.

Essa função é a seguinte:

```
gluLookAt( ex, ey, ez, cx, cy, cz, ux, uy, uz );
```

Todos os argumentos são `doubles`. `ex`, `ey` e `ez` definem as coordenadas da câmara (ou olho ou *eye)*, `cx`, `cy` e `cz` definem as coordenadas para onde estamos a olhar (centro de foco) e `ux`, `uy` e `uz` definem um vector que indica para onde é "cima" *(up)*, de forma a especificarmos a orientação em termos de rotação.

Animação: *double-buffering*

Como já foi discutido, animar uma cena é desenhar vários fotogramas com pequenas alterações que indicam movimento ou outras alterações.

Para simular movimento, podemos simplesmente redesenhar exactamente a mesma cena, com os mesmos objectos, modificando apenas a translação dada a um ou mais objectos.

Mas redesenhar uma cena pode causar problemas. Antes de começar a desenhar devemos limpar o *viewport*, começando depois a enviar polígonos para o OpenGL. Por muito rápido que este desenho seja, pode existir um instante em que nos apercebemos de ver um fotograma parcialmente desenhado. Conforme isto acontece num e noutro fotograma, a imagem parece "piscar" *(flicker)*.

Existem duas tecnologias que evitam este efeito. A primeira é o *double-buffering*. Se a placa gráfica tiver memória suficiente e o suportar, ela tem duas zonas de memória *(buffers)* para cada *viewport*, guardadas em memória. Uma é a que está neste momento a ser desenhada no ecrã. A outra é onde os comandos OpenGL estão a desenhar os polígonos. Quando esse desenho termina, a placa gráfica troca os *buffers* instantaneamente, sem nunca ser visível uma cena "inacabada". São normalmente os sistemas subjacentes que fazem essa troca automaticamente, o que é o caso do Qt.

Outra tecnologia é a sincronização da troca de *buffers* com o sinal *vertical-retrace* do monitor. Isto garante que a troca não ocorre durante o desenho do *viewport* no ecrã, fazendo com que um dos fotogramas tenha visível a metade superior do *buffer* antigo e a metade inferior do novo *buffer*, efeito se ocorresse com frequência tornava-se visível. Esta tecnologia está sempre activa em OpenGL.

Estas duas tecnologias garantem animação suave em OpenGL, mas a segunda impõe um limite na animação: a quantidade de fotogramas por segundo (fps). Como a troca de *buffer* está sincronizada com o início de um novo fotograma, não conseguimos exibir fotogramas com mais frequência do que aquela que o monitor suporta (geralmente por volta de 50 a 60 fps).

Animação: temporização

Precisamos então de fazer duas coisas para fazer uma animação:

1. Uma forma de a cada **d** milissegundos* repetir nova cena, e
2. Uma forma de saber que alterações fazer em cada nova cena.

O primeiro requisito tem de ser satisfeito pelo sistema que está a suportar a aplicação. Não faz parte da norma OpenGL lidar com temporizações.

Vê nos exercícios abaixo como é que o Qt resolve a questão de temporizações, na prática.

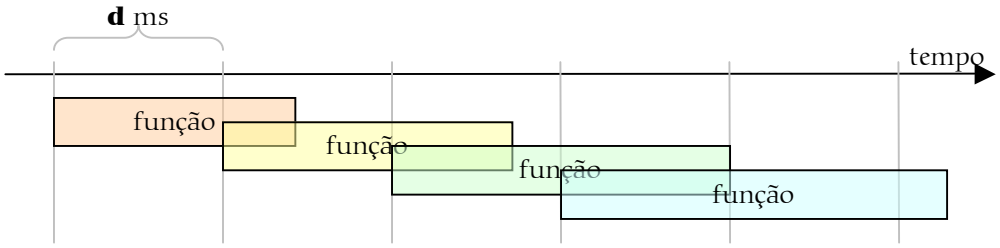
O conceito em si é semelhante em todos os sistemas: fazer com que uma função da minha aplicação seja chamada a cada **d** ms. Mas como escolher o **d**?

Depende do sistema e da suavidade pretendida para a animação. Em alguns sistemas, o "relógio" que controla as várias chamadas à minha função começa a contar um novo período de **d** ms assim que chama a minha função. Se a minha função ainda estiver a correr quando voltarem a passar **d** ms, ocorrem uma de três coisas:

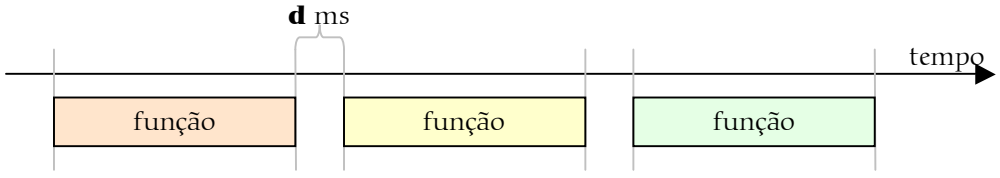
1. Ele espera que a minha função termine antes de a voltar a chamar;
2. Ele chama de imediato a minha função;
3. Ele ignora esta oportunidade de chamar a minha função.

Seja qual for o caso, ele continua imediatamente a contar novo período de **d** ms.

Repara que se estiveres a trabalhar num sistema que funcione como 1 ou 2, podes sobrecarregá-lo se a tua função não terminar antes de **d** ms. Cada nova chamada à tua função é mais lenta porque corre ao mesmo tempo que a anterior, e assim por diante:



No Qt, o intervalo especificado é o lapso de tempo entre duas chamadas à minha função pelo que estes problemas não ocorrem:



* Milésimos (1/1000) de segundo, abreviado "ms".

Lembra-te que a tua aplicação pode correr em computadores bastante piores do que o teu pelo que experimentar se a tua função consegue fazer tudo em **d** ms, não é garantia que sempre o consiga.

A suavidade com que eu quero fazer a animação é o que determina o **d**. Segundo alguns autores, o olho humano precisa de ver 16 fotogramas por segundo ou mais para os entender como imagens em movimento (i.e., sem se aperceber dos fotogramas individuais). Então precisamos de um **d** de 1/16 segundos ou 1000/16 ≈ 62 milissegundos, no máximo. Por outro lado, uma frequência acima dos 60 fotogramas por segundo está acima daquilo que a maior parte do hardware consegue reproduzir, pelo que o **d** mínimo é 1000/60 ≈ 16 milissegundos.

O **d** ideal estará entre estes valores e depende da vossa aplicação e necessidades de CPU vs. suavidade na animação.

Então e agora como sei que alterações fazer em cada nova cena? Se o tempo que a minha função de desenho da cena demora varia bastante de computador para computador, como é que eu garanto que o movimento de um objecto na cena do ponto **A** para o ponto **B** que deveria demorar 1s, demora de facto 1s?

A única forma de garantir isto é a tua função ler o relógio do sistema. Quando esse objecto iniciar o seu movimento, deve ser lido o relógio (com precisão de milissegundo). Cada vez que vou desenhar nova cena, leio de novo o relógio para saber quanto tempo passou e faço algumas contas para saber onde deveria estar o objecto neste momento para o desenhar nesse sítio.

O intervalo **d** neste caso serve apenas para impedir que o sistema fique com carga a 100% sem necessidade disso e que os processos normais (habitualmente de interface com o utilizador ou actualização das regras do jogo) continuam a correr.

Exercícios

Imagina um quadrado desenhado em coordenadas 3D, mas no plano $z=0$, e com os cantos (x,y,z):

- (-1,1,0)
- (1,1,0)
- (1,-1,0)
- (-1,-1,0)

Imagina que quero rodar o quadrado ao longo do eixo dos x, 90° (ou seja, se o quadrado fosse o ecrã do teu portátil, estou a pegar na parte de cima e a deslocá-lo para baixo até ficar paralelo com a mesa: lembra-te no entanto que o eixo de rotação do ecrã é diferente do deste quadrado). Quero também depois deslocá-lo para longe de mim (z negativo), por 1 unidade.

- Faz um esboço em papel do quadrado inicial e sua posição final.

- Faz as matrizes de transformação.
- Faz os cálculos separados com cada matriz.
- Combina as duas matrizes e faz os cálculos com a matriz resultante.
- Vê se ambos os resultados batem certo.

Compila e corre o exercício NeHe ch4.

- No ficheiro `nehewidget.cpp`, função `NeHeWidget::NeHeWidget()` vês o código que cria o temporizador. Ele chama a função `timeOut()` do ficheiro `ch4.h` a cada 50ms (vê a linha 8 do ficheiro `ch4.h`).
- Reduz o lapso de 50ms para 20ms e vê o que acontece. É o esperado?
- Estuda as funções `paintGL()` e `timeOut()` do ficheiro `ch4.h`. Mantendo o lapso em 50ms, tenta acelerar a rotação das figuras alterando apenas o código em `timeOut()`.
- Faz com que as figuras rodem ambas em torno do eixo dos *y* central, enquanto rodam nos seus próprios eixos. Repara: eu desenho sempre o triângulo primeiro e depois o quadrado, pelo que quando eles se cruzam, deve estar sempre o quadrado "em cima" (ou "à frente") do triângulo, certo? Verificas isto? Que conclusões tiras?
- Altera o código para passares a pensar de forma diferente. Em vez de rodares as figuras, faz uma câmara rodar à volta delas (usa `gluLookAt()`).

Aula 04

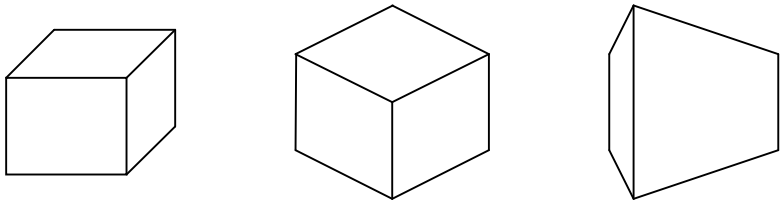
Projecções ortogonais e perspectivas. Relação de aspecto.

O jogo CGMaze: desenhar o mapa com operações OpenGL 2D.

Projecções

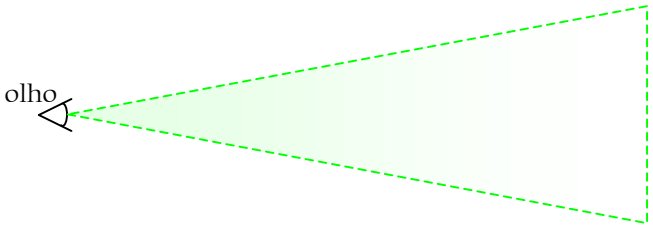
Projecções são formas de representar uma cena 3D num plano 2D. Repara que o ecrã de um computador tem apenas duas dimensões (2D): altura e largura. Como é que crio então a ilusão de profundidade?

Há várias formas de o fazer. Vê por exemplo estes cubos:



O ecrã ou folha de papel onde estás a ler isto é plano (2D), mas no entanto estas linhas oblíquas dão-te a ilusão de um objecto tridimensional. A este tipo de projecção chama-se **perspectiva**.

Ela imita a visão humana da realidade. Se, muito perto da cara, colocares o polegar e indicador muito afastados, não vais conseguir ver a ponta desses dedos. Mas se esticares o braço, não os consegues afastar o suficiente para os deixares de ver.



Isto acontece porque a lente esférica dos nossos olhos apanha luz que se situa dentro de um cone à sua frente.

Por outro lado, se eu alinhasse a minha visão para estar exactamente de frente para o cubo, tudo o que veria seria isto:

Se a face de trás tivesse exactamente as mesmas dimensões quando desenhada no papel e as arestas que as ligam estivem paralelas ao eixo imaginário dos *z* que sai do ecrã, então chamo a esta projecção uma projecção paralela ou **ortogonal**[*].

Claro que no mundo real, mesmo alinhado de frente para o cubo, a face de trás do mesmo seria mais pequena[†]. A projecção ortogonal não imita a visão humana da realidade, mas é muito usada em planos técnicos de engenharia (arquitectura, aeroespacial, etc.).

Projecções em OpenGL

Eu crio uma projecção de perspectiva em OpenGL em que o olho está na origem das coordenadas a olhar na direcção do eixo negativo dos *z*, com:

```
gluPerspective( cvy, aspecto, zperto, zdistante );
```

onde todos os argumentos são `doubles` e `cvy` é ângulo do campo de visão medido em relação ao eixo dos *y*, `aspecto` é a relação de aspecto (normalmente do *viewport)* medida por

$$aspecto = \frac{largura}{altura}$$

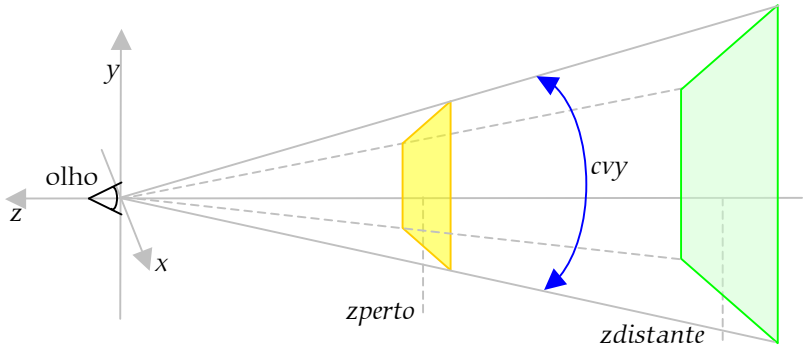
e `zperto` e `zdistante` são distâncias a partir da origem e na direcção do eixo negativo dos *z*. Só são desenhados pontos que estão entre `zperto` e `zdistante`[‡]. `zperto` também define o plano de projecção que representa o ecrã.

[*] Ou ainda "ortográfica".

[†] Experimenta olhar de cima para um copo que tenha os lados direitos. Irás conseguir ver o fundo do copo completo, "dentro" do círculo que define a boca do copo.

[‡] Ou seja, elas definem os planos de corte perto e distante *(near and far clipping planes)*: mais sobre isto mais tarde.

Como o ecrã é rectangular (e não circular, como o olho humano), a perspectiva define uma pirâmide quadrangular em vez de um cone.



Eu crio uma projecção ortogonal com:

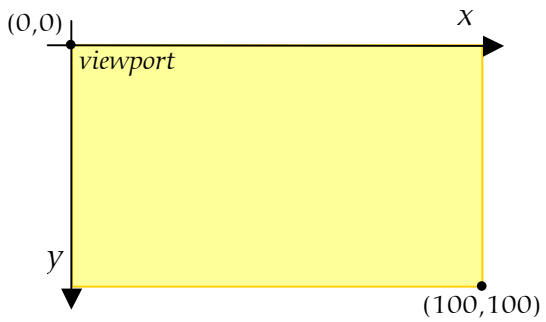
```
gluOrtho2D( esquerda, direita, baixo, cima );
```

onde todos os argumentos são `doubles` e representam as coordenadas limítrofes no *viewport*.

Por exemplo, se usar:

```
gluOrtho2D( 0, 100, 100, 0 );
```

então vou estar a trabalhar num *viewport* cujo sistema de coordenadas é:



No entanto nota que devo usar as funções OpenGL 2D (i.e., com $z=0$) nesta projecção. Para poder usar outros valores de z, consulta a função `glOrtho()` no manual OpenGL.

Matrizes de projecção

Os cálculos de projecção (em perspectiva ou ortogonal), podem ser feitos com matrizes adequadas. De facto, as funções OpenGL explicadas nesta aula, multiplicam a matriz de projecção actual pela matriz gerada pela função.

Mas a matriz de transformação que estudámos na aula anterior é aplicada separadamente desta. Como indicamos então que matriz queremos modificar?

Para escolher a matriz de projecção, usamos a função:

```
glMatrixMode( GL_PROJECTION );
```

E para escolher a matriz de transformações normal, usamos:

```
glMatrixMode( GL_MODELVIEW );
```

Esta função também aceita como argumento `GL_TEXTURE` para outra matriz.

Todas as operações seguintes sobre matrizes (carregar identidade, transformações e projecções) ocorrem sobre a matriz seleccionada por esta função.

Exercícios

Porque é que nos exercícios da aula 2 foi necessário adicionar uma linha `glTranslatef(0,0,-10)`?

Porque é que nos exercícios da aula anterior é necessário que o `glTranslatef()` tenha uma translação *z* de `-6.0`?

Altera o exercício da aula 2 (ch2) para passar a ter uma projecção ortogonal em vez de perspectiva.

Abre o projecto "CGMaze". Abre os ficheiros `student_viewmap.h` e `student_viewmap.cpp`. Usando projecções ortogonais e OpenGL 2D, completa esses ficheiros. Dicas:

- Depois de localizares quais funções vais ter de preencher, começa por idealizar em papel qual o sistema de coordenadas que vais usar (vais precisar disso para desenhar os quadrados e para estabelecer o sistema de coordenadas com `gluOrtho2D()`). Onde vai estar a origem? Provavelmente no mesmo sítio da origem das posições `x`, `y` que o método `ViewMap::paint(int, int, Cell)` vai receber. As posições que esse método recebe estão no mesmo sistema de coordenadas idealizado no ficheiro `map.h`. Agora que coordenadas visuais (de OpenGL) vão ter os vértices do quadrado que vai ser desenhado para a célula (`Cell`) na posição 0,0?[*] E a célula na posição 0,1? Com que contas algébricas transformo as coordenadas das células nas coordenadas dos vértices do quadrado respectivo?
- Começa por preencher o método `ViewMap::paint(int, int, Cell)` antes do `ViewMap::paint()`. O primeiro desenha um quadrado de cada vez, enquanto o segundo desenha todo o mapa "que foi desenhado até agora". Deixa o segundo para último, depois de preencheres os restantes métodos.
- Antes do `glBegin()` que vais criar para desenhar cada quadrado, estabeleces a cor de desenho para parede, chão e jogador 1 respectivamente com:

```
glColor3ub( VIEWMAP_COLOR_3UB_WALL );
```

[*] Isso depende de que altura e largura queres dar ao teu quadrado. Determina isso arbitrariamente.

```
glColor3ub( VIEWMAP_COLOR_3UB_FLOOR );
glColor3ub( VIEWMAP_COLOR_3UB_PLAYER1 );
```

Recorda os exercícios da aula 1 para saber como determinar o que a célula passada ao método `ViewMap::paint(int, int, Cell)` contém.

- Para o método `ViewMap::paint(int, int, Cell)` desenhar os quadrados de imediato, assegura-te que ele termina com uma chamada a `glFlush()`. Esta função não é normalmente usada excepto para algumas situações de animação.
- Apagas visualmente o *viewport*, (que deves (re)estabelecer em `ViewMap::resize()` – relembra a aula 2 e o exercício ch2) com:

```
glClear( GL_COLOR_BUFFER_BIT );
```

- Inicializa o OpenGL (no construtor da classe) com:

```
glShadeModel( GL_SMOOTH );
glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );
glDisable( GL_DEPTH_TEST );
```

  É nessa inicialização que também deves ter o teu `gluOrtho2D()`, com as funções auxiliares `glMatrixMode()` e `glLoadIdentity()` à volta (vê exemplo ch2 que acabaste de fazer no exercício anterior a este).
- Neste momento o mapa deve estar a ser desenhado correctamente.
- Preenche por fim o método `ViewMap::paint()`. Para este saber o que desenhar, deve "lembrar-se" do que vai sendo enviado para `ViewMap::paint(int, int, Cell)`. Cria então uma variável que guarde estas células, para depois a varreres e enviares para o ecrã chamando este último método dentro de dois ciclos `for()` encadeados (vê comentário já existente em `ViewMap::paint()` para um exemplo). Nota que o `glFlush()` ao final de `ViewMap::paint(int, int, Cell)` faz com que este processo seja lento: tenta optimizá-lo.

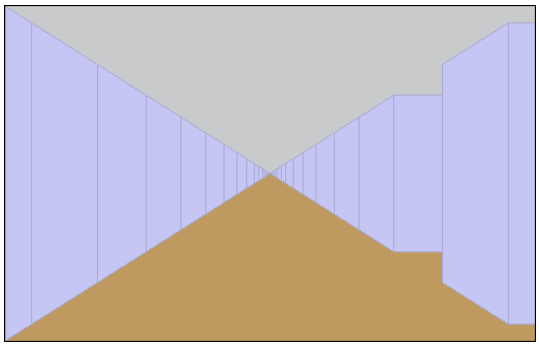
Aula 05

O jogo CGMaze: desenhar a vista interior com operações OpenGL 2D.

Exercício: Desenhar a vista interior do labirinto

Vamos desenhar a vista interior do labirinto usando operações OpenGL 2D. Este exercício visa familiarizar-te com alguns conceitos e dar-te uma base de comparação para vires a ter a noção da simplicidade que na realidade é desenhar objectos com OpenGL 3D em vez de usar operações 2D.

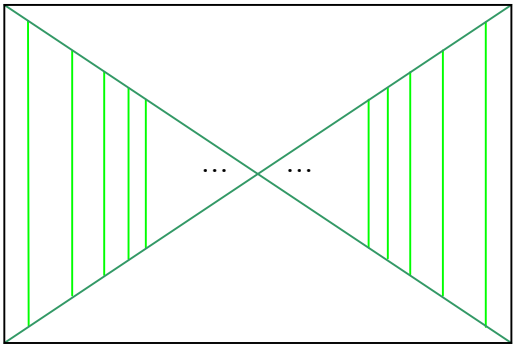
Repara nesta vista do labirinto:



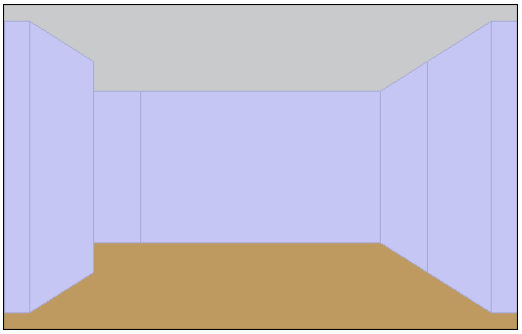
O "céu" é sempre cinzento e o "chão" é sempre castanho. Então é começo a desenhar a vista do interior do labirinto desenhando o horizonte:

Agora desenho as paredes do corredor que está à frente do jogador. Como faço isso sem recorrer a OpenGL 3D?

Neste caso (labirinto) posso simplificar o processo. Repara como as paredes são sempre desenhadas com duas linhas cruzadas em X e linhas verticais entre elas a fazer de divisórias entre cada parede.

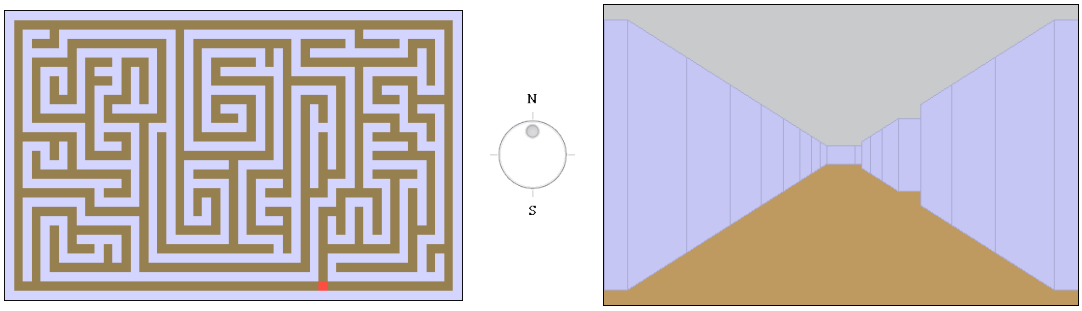


Mas não devo desenhar de imediato as linhas cruzadas, porque elas são interrompidas quando encontro uma parede de frente. Vê este exemplo:



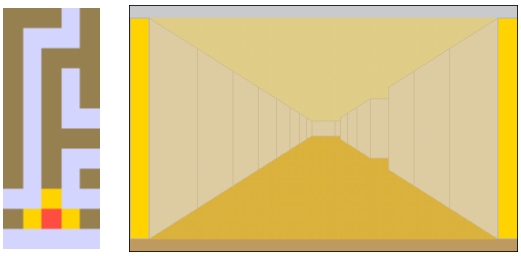
Isto significa que tenho de ir desenhando as paredes, das que estão mais perto do jogador para as que estão mais longe, verificando sempre se encontro uma parede de frente. Quando encontrar, desenho-a e paro o processo de desenho.

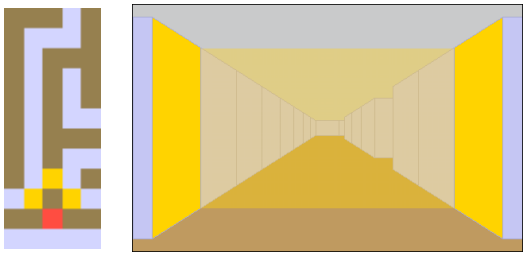
Por exemplo, imagina que quero desenhar esta vista do labirinto:

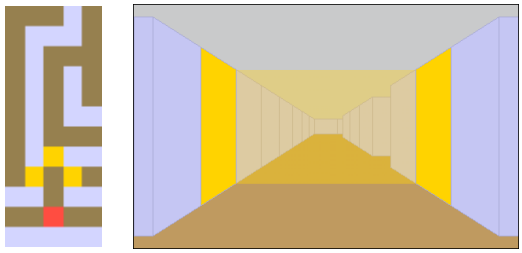


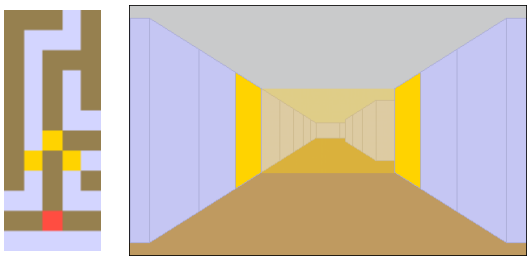
Eu preciso de avançar, a partir da posição do jogador e na direcção em que ele está a olhar (Norte, neste caso), até encontrar uma parede de frente.

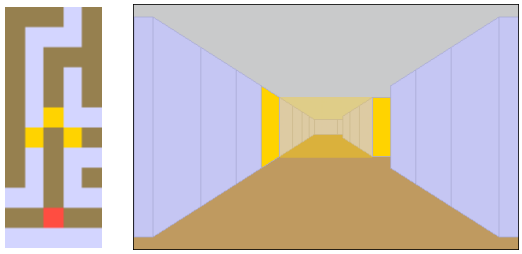
O processo é o seguinte:

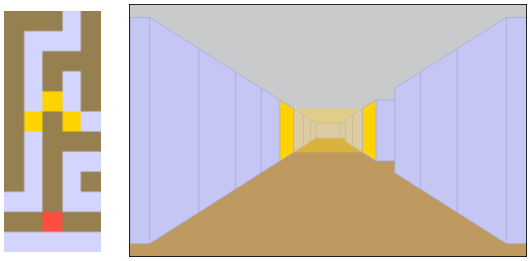
1. 

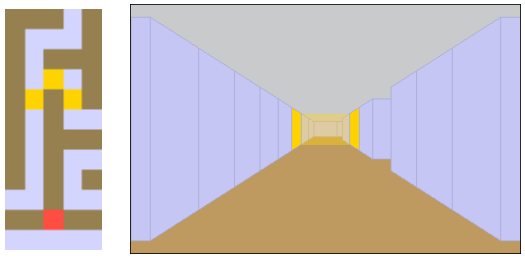
2. 

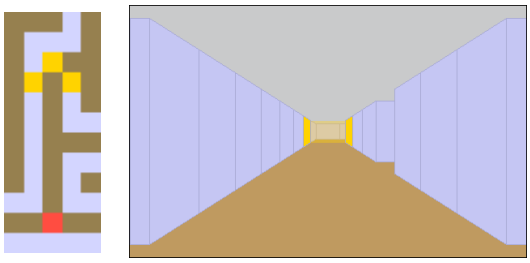
3. 

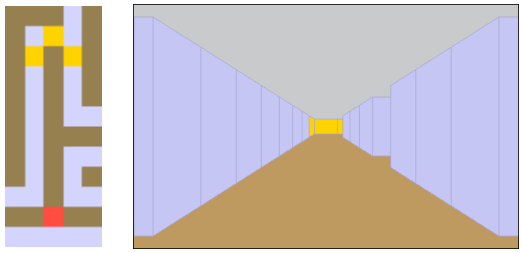
4. 

5. 

6. 

7. 

8. 

9. 

Em cada passo, desenhar a parede de frente é trivial (é um rectângulo). As paredes ou corredores laterais também são extremamente simples, já que são trapézios (parede) ou rectângulos (corredor).

Se eu estabelecer um sistema de coordenadas quadrado (que será distorcido para o rectângulo do *viewport* pelo OpenGL), desenhar as linhas obliquas dos trapézios é extremamente simples, uma vez que serão diagonais a 45°.

Se tiver nas variáveis `x` e `y` as coordenadas do mapa do jogador e em `compass` a sua bússola (orientação), então o seguinte código percorre o corredor em frente ao jogador:

```
for( n = 0;  n < VIEW2D_DEPTH;  n++ )
      {
      map->moveLeft ( &x, &y, compass );
      // posicao da esquerda
      if( map->isWallOrDoor(x, y) )
            // desenha parede
      else
            // desenha corredor
```

```
            map->moveRight( &x, &y, compass );
            map->moveRight( &x, &y, compass );
            // posicao da direita
            map->moveLeft ( &x, &y, compass );
            map->moveFwd  ( &x, &y, compass );
            // posicao em frente ao jogador
            // lembra-te de fazer break se for parede!
            }
```

Aqui, `VIEW2D_DEPTH` é uma constante que define a profundidade máxima de desenho. O valor de `100` é uma boa aposta para esta constante, mas depende de como vais diminuindo a largura dos trapézios desenhados.

Vamos então pôr isto à prática.

Abre o projecto "CGMaze". Abre os ficheiros `student_view2d.h` e `student_view2d.cpp`. Usando projecções ortogonais e OpenGL 2D, completa esses ficheiros. Dicas:

- As funções `View2D::View2D()` e `View2D::resize()` vão-se manter muito semelhantes ao que já fizeste para o desenho do mapa.
- As macros que começam com `VIEW2D_COLOR_3UB_` definidas em `student_view2d.h` devem ser usadas para cores, tal como no desenho do mapa.
- A função `View2D::paint()` é chamada para desenhar a vista do jogador na posição do mapa `x` e `y`, orientado para `compass`. Ela terá então o código descrito anteriormente para percorrer o corredor em frente ao jogador, assim como código teu para desenhar o horizonte e as paredes e corredores.

Aula 06

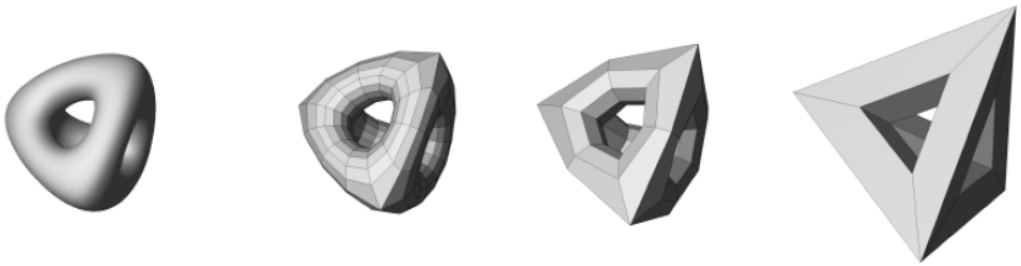
Representação de superfícies e objectos sólidos *(polygon meshes*).

Cálculo de visibilidade (superfícies escondidas).

Transparência (canais alfa, *blending* e *depth buffer)*.

Representação de objectos sólidos

Um objecto sólido é representado aproximando a sua superfície com triângulos e/ou quadriláteros (ou polígonos em geral). Quando mais densa a rede de pontos que define essas superfícies, melhor aproximado fica o objecto.



Em Inglês, chama-se a esta rede de polígonos, uma *polygon mesh*.

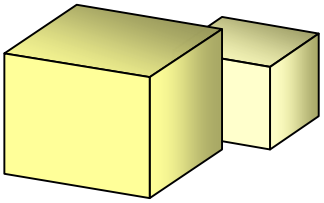
Desenhamos cada polígono com os habituais `glBegin()` e `glEnd()`. O objecto deverá estar representado na nossa aplicação em algum formato que seja prático (vector, lista, etc.).

Cálculo de visibilidade

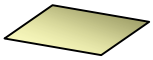
Existem duas situações em que uma face ou superfície pode não estar visível:

1. Se está total ou parcialmente oculta por outra superfície, ou
2. Se está "de costas" para o observador.

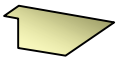
A primeira situação é resolvida com recurso à medição da distância entre o observador e a superfície. Superfícies mais perto ocultam superfícies mais distantes.



Repara como o cubo mais perto oculta parcialmente algumas faces do cubo mais distante. Repara ainda como a face de cima do cubo de trás deixa de ter a forma



para ter a forma



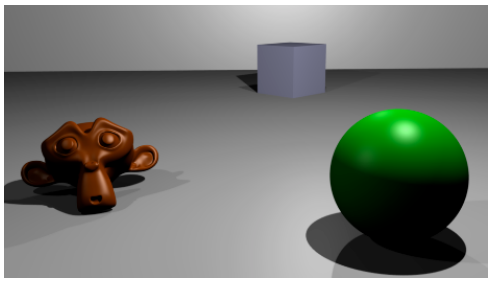
Não é trivial determinar o aspecto de uma superfície após a sua ocultação parcial.

Uma das técnicas usadas por motores 3D para resolver o problema da ocultação de superfícies é ordená-las por distância ao observador e desenhá-las das mais distantes para as mais próximas. A esta técnica chama-se de *z-ordering*.

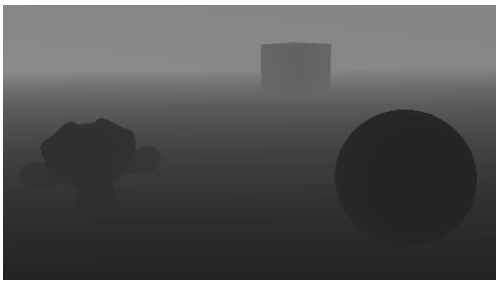
Esta técnica tem duas vantagens:

1. Por mais complexa que seja a ocultação parcial de superfícies (lembra-te que cada superfície pode ser parcialmente oculta por várias outras superfícies) a técnica resulta num aspecto correcto,
2. Permite o desenho e uso de superfícies semi-transparentes.

Outra técnica é de cada vez que a placa gráfica desenha um *pixel* de uma qualquer superfície, guardar a distância a que ele está do observador. Se o *pixel* que vai desenhar está mais próximo do que aquele que já existe no ecrã na mesma posição, substitui-o, caso contrário, ignora-o. A esta técnica chama-se de *z-buffering* ou *depth-buffering*.



Cena tridimensional*.



O *depth-buffer* respectivo.

A vantagem desta técnica é que se duas superfícies se cruzam, ela representa-as de forma realística.

* Imagem retirada da Wikipedia; autor "Zeus".

Activo o *depth-buffer* em OpenGL com:

```
glDepthFunc( GL_LEQUAL );
glEnable( GL_DEPTH_TEST );
```

A função `glDepthFunc()` não vai ser explicada. Consulta o manual do OpenGL para mais informações.

Desactivo-o com:

```
glDisable( GL_DEPTH_TEST );
```

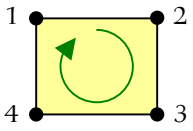
Como isto cria um novo *buffer* (o *depth-buffer*), tenho de me lembrar de agora em diante, quando quiser apagar o "ecrã" do OpenGL, usar:

```
glClear( GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT );
```

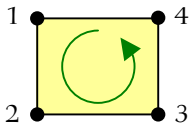
Estas técnicas perdem no entanto tempo precioso quando acaba por desenhar superfícies que estão totalmente ocultas.

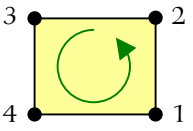
Por fim, repara como nos cubos que estão desenhados acima metade das suas faces estão totalmente ocultas. Essas são sempre também as faces que estão "de costas" para o observador. A eliminação das faces de trás dos objectos (em Inglês, *back face culling*), é assim outra técnica que pode poupar à placa gráfica até 50% do tempo de desenho de um objecto.

Em OpenGL eu faço essa eliminação ao indicar os pontos de cada superfície por uma ordem específica quando olho para ela pelo seu lado visível (a "frente"). Se quando projectada, esta superfície tiver os seus pontos pela ordem oposta, ela está "de costas" para o observador.



Ordem dos ponteiros do relógio





Ordem inversa aos ponteiros do relógio

Eu posso indicar que as faces da frente são indicadas com pontos na ordem inversa à dos ponteiros do relógio *(counter clockwise)*, que é o valor por omissão, com:

```
glFrontFace( GL_CCW );
```

O inverso (ordem dos ponteiros do relógio, ou *clockwise*), faz-se com:

```
glFrontFace( GL_CW );
```

Para que o OpenGL elimine agora as faces dos polígonos com base na ordem pela qual eu lhe dou os pontos, faço:

```
glCullFace( GL_BACK );
glEnable( GL_CULL_FACE );
```

O argumento para `glCullFace()` pode também ser `GL_FRONT` (para ele não desenhar as faces que estão de frente para o observador) ou

`GL_FRONT_AND_BACK` para ele desenhar apenas pontos e linhas, mas nenhuma face.

Transparência e opacidade

Transparência (ou, inversamente, opacidade), é uma característica de superfícies que se aplica em desenho 2D e 3D. Água, óculos, copos, janelas e faróis são exemplos de alguns objectos que são parcialmente transparentes. Mar, óculos escuros e vidros fumados são exemplos de objectos com níveis maiores de opacidade (i.e., menos transparentes).

O grau de transparência de uma imagem (foto) pode variar dentro da imagem. Uma imagem tem dimensões quadrangulares, mas se eu tiver zonas na imagem com transparência de 100%, consigo ter imagens com qualquer formato. Para poder ter uma transparência diferente em cada *pixel* que compõe a imagem, cada *pixel* tem então de ter informação sobre a sua transparência (ou melhor, opacidade), para além da sua cor. Chama-se a essa informação de opacidade o "canal alfa" e como complementa os três canais de cor (RGB[*]) que definem a cor do *pixel*, chama-se habitualmente a uma imagem neste formato, uma imagem RGBA.

Esta funcionalidade (chamada de *blending* em OpenGL) tem de ser activada pois o seu cálculo desacelera o processamento normal. Activo-a com:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable( GL_BLEND );
```

e desactivo-a com:

```
glDisable( GL_BLEND );
```

A função `glBlendFunc()` não vai ser explicada. Consulta o manual do OpenGL para mais informações.

Posso usar o canal alfa em imagens dadas como texturas ao OpenGL ou como um quarto valor numérico quando especifico uma cor. Mais detalhes serão dados em aula posterior, quando cobrirmos estes temas.

[*] *Red*, *Green*, *Blue*. Mais sobre isto em aula posterior.

Exercícios

Compila e corre o exercício NeHe ch5. Este desenha dois objectos sólidos tridimensionais: uma pirâmide de base quadrangular e um cubo.

- Localiza o código que desenha ambos os sólidos.
- O código que desenha a pirâmide usa apenas superfícies triangulares. A superfície inferior da pirâmide devia no entanto ser um quadrado. Olha para o código e explica o que se está a passar. Altera o código para verificares a tua hipótese.

Compila e corre o exercício NeHe ch8. Quando correres este programa podes usar as setas para acelerar o cubo nos seus eixos, PgUp e PgDown para ampliar/reduzir, L para activar/desactivar a iluminação *(lighting* – mais sobre isto em aula posterior), B para activar/desactivar a transparência *(blending)* e F para iterar por 3 filtros (para qualidade da imagem – mais sobre isto em aula posterior).

- Experimenta activar a transparência (B) sem rodar o cubo. Repara como vês a face de trás do cubo mais pequena do que a face da frente, tal como vimos na aula sobre perspectivas.
- Experimenta activar a transparência (B) e roda o cubo (setas). Repara como as faces são todas desenhadas, mesmo se as estamos a ver "por detrás". Como sabe o OpenGL quando desenhar as faces apesar de estarem a ser vistas "por detrás"? Corrige isto. Vê como o resultado não é o desejado, quando temos faces semi-transparentes.
- Altera a coordenada z do 1º vector para passar a ser -1. Corre de novo o programa e activa iluminação (L) e transparência (B). Repara no resultado.

Aproveita o tempo que sobrar nesta aula para completar o exercício da aula anterior.

Aula 07

Cor (interpretação da cor, modelos de cor, cones de cor nos olhos; emissores de cor nos *pixels)*.

Profundidade de cor: P/B, escala de cinzentos, *palette* ou RGB.

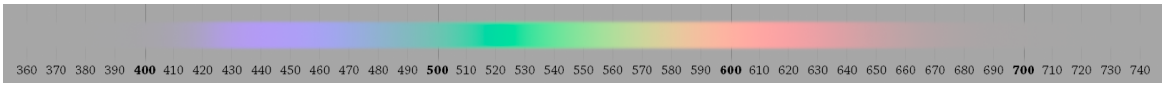
Tipos de ficheiros de imagens (GIF, TIFF, PNG, JPG); Compressão *lossy* vs. *lossless*. Canais alfa.

Texturas.

O que é a cor

Se estiveres fechado no teu quarto à noite, sem fontes de luz, não vês nada. A "cor" predominante é o preto. Se apontares uma *Webcam* para o Sol (nunca faças isso com os olhos, sem protecção), ela "satura" e mostra uma imagem branca. Cor está então relacionada com luz. Quando de um objecto não chega luz aos nossos olhos, o objecto parece-nos preto. Se chega muita luz com muita intensidade, ele parece-nos branco.

Mas para além da intensidade da luz, ela tem outra propriedade que é a sua frequência. A luz é radiação electromagnética (o mesmo tipo de radiação usada em rádio, WiFi, telefones celulares, televisão por antena, etc.) e é "feita" de fotões, partículas que vibram com uma rapidez (frequência) variável. Quando essa frequência estiver aproximadamente entre os 400THz[*] e os 700THz, essa luz é visível e tem cores diferentes dependendo da frequência.



Espectro de Luz[†]

Se os nossos olhos virem um objecto que emite luz nas frequências verde e encarnada, esse objecto é visto como tendo cor amarela (a combinação das duas). A combinação de todas as cores é visto como cor branca, e a sua ausência como cor negra.

Cada objecto reflecte (ou emite) para os nossos olhos uma ou mais destas frequências. A cor que vemos num objecto tem a ver com a combinação destas frequências.

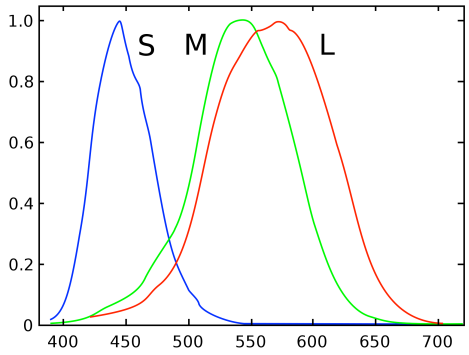
---

[*] Tera-Hertz. 1Hz é um ciclo por segundo. 1T são 1000G (mil gigas), ou 10^{12} (um bilião). 1THz são é então um bilião de ciclos por segundo. Vê na bibliografia, [SI].

[†] Imagem retirada da Wikipedia; autor "Spigget".

Percepção de cor pelo olho humano

A retina no olho humano tem três tipos de células sensíveis à luz. São chamados de cones de cor. Um desses tipos é apenas sensível à luz violeta *(short wavelength*, **S**), o outro à luz verde *(middle wavelength*, **M**) e o terceiro à luz amarela *(long wavelength*, **L**).



Curva de resposta dos cones em relação às frequências de luz

Existe um quarto tipo de células que são os bastonetes, mas estes apenas estão activos com luz de baixa intensidade. Como só existe um tipo destas células, o ser humano vê a "preto-e-branco" em situações de pouca luz.

Geração de cor

Thomas Young propôs em 1801 a sua teoria tricolor, ao observar que qualquer cor podia ser criada usando apenas três luzes de cores específicas. James Clerk Maxwell e Hermann von Helmholtz mais tarde refinaram esta teoria.

Existem dois grandes modelos tricolores usados:

- RGB, sigla das cores encarnado *(Red)*, verde *(Green)* e azul-mar *(Blue)*. Estas são as chamadas cores aditivas, pois adicionam cor a um preto base. Como tal, este é o modelo usado em ecrãs de televisão e computador de todos os tipos (CRT, LCD, LED, etc.).
- CMYK, sigla das cores azul-céu *(Cyan)*, magenta *(Magenta)*, amarelo *(Yellow)* e preto *(blacK)*. Estas são as chamadas cores subtractivas (excepto o preto), pois fazem com que a superfície onde são aplicadas (geralmente uma folha de papel branco) deixe de reflectir todas as frequências de luz. Como tal, este é o modelo usado em impressoras de todos os tipos. A cor preta tem a ver com conseguir um preto mais saturado do que o conseguido pela combinação das outras três cores, o que é extremamente importante para a legibilidade de texto em papel.

É importante entender que as cores que não são as cores principais do modelo, nunca serão tão saturadas (brilhantes, "vivas") quanto uma luz que emita directamente nessa cor. Existem assim limites cromáticos nestes modelos, mas estes ainda aproximam suficientemente bem essas cores.

Cor em OpenGL

Em Computação Gráfica interessa-nos obviamente o modelo RGB. Já te foi mostrado na aula 02 uma ampliação do ecrã de um computador com os seus *subpixels* (emissores de cada uma das cores RGB em cada *pixel*). Este é então o modelo usado pelo OpenGL.

Escolho uma cor em OpenGL com uma das seguintes funções:

```
glColor3f( r, g, b );
```

onde `r`, `g` e `b` são `floats` entre 0.0 (essa cor está desligada) e 1.0 (essa cor está na sua intensidade máxima), ou:

```
glColor3ub( r, g, b );
```

onde `r`, `g` e `b` são bytes sem sinal entre 0 (desligada) e 255 (intensidade máxima).

Existem várias outras variantes destas funções com outros tipos de dados. Consulta o manual do OpenGL para mais informações.

Estas funções devem ser usadas antes de especificar o ponto que terá a dita cor dentro de um bloco `glBegin()` a `glEnd()`, embora possam também ser usadas antes desses blocos.

Como especificam cores de pontos, estas funções podem ser usadas para também especificarem a transparência desses pontos. Para isso substituímos o `3` no nome da função por um `4`, e adicionamos um último (4º) argumento que será a opacidade desse ponto (0 para total transparência).

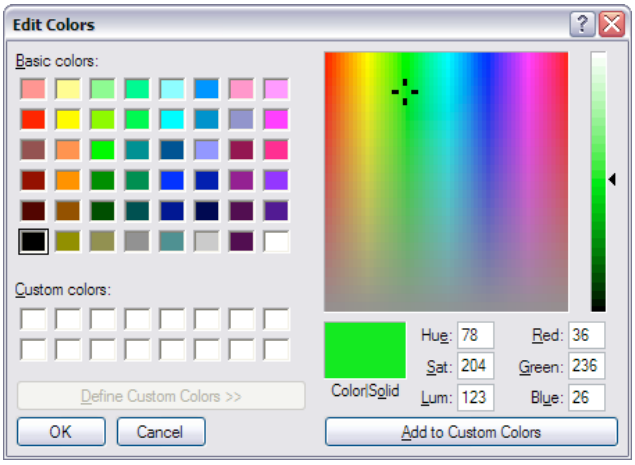
A função:

```
glClearColor( r, g, b, a );
```

onde `r`, `g`, `b` e `a` são `floats` entre 0.0 e 1.0, especifica qual a cor e opacidade (alfa) do *viewport* quando o quisermos apagar com `glClear()`.

Como escolho os valores RGB para uma cor? Em praticamente qualquer aplicação que suporte cor, podes escolher visualmente uma cor e ela dá-te os valores RGB (habitualmente em bytes sem sinal) para essa cor.

Por exemplo, na aplicação Paint dos acessórios do Windows posso editar as cores:



Como vês no canto inferior direito, a cor que escolhi tem os valores r=36, g=236 e b=26 (bytes sem sinal), e poderia ser escolhida em OpenGL com:

```
glColor3ub( 36, 236, 26 );
```

Texturas

A forma mais simples que OpenGL usa para simular uma textura numa superfície é simplesmente "pintar" essa superfície (que geralmente é um quadrilátero) com uma imagem.

O OpenGL precisa de receber a imagem num formato especial, pelo que precisa sempre da ajuda do sistema onde está a correr. No Qt, abrimos essas imagens com:

```
QImage i, igl;

if( !i.load("imagem.png") )
      {
      i = QImage( 16, 16, QImage::Format_RGB888 );
      i.fill( QColor(Qt::green).rgb() );
      }
igl = QGLWidget::convertToGLFormat( i );
```

As partes mais claras dependem da tua aplicação. O que estamos a fazer é tentar abrir um ficheiro de imagem. Se não o conseguirmos abrir, entramos no `if()` e criamos uma imagem de 16×16 *pixels* sólida de cor verde (ou outra que prefiras), de forma a assegurar que a aplicação ainda funciona mesmo que não tenha a imagem disponível.

Repetimos este processo para todas as imagens que precisarmos.

Criamos agora no OpenGL tantos identificadores de textura (um identificador é um número inteiro que o OpenGL te irá dar) quantos as texturas (imagens) que vou usar (neste caso, 1):

```
GLuint texturas[1];

glGenTextures( 1, texturas );
```

Para cada textura (0 é a primeira e única neste caso), vou agora carregar e configurá-la no OpenGL. Começo por escolher a textura que vou configurar:

```
glBindTexture( GL_TEXTURE_2D, texturas[0] );
```

`glBindTexture()` escolhe uma das texturas criadas com `glGenTextures()` para ser usada nas funções OpenGL posteriores relativas a texturas. Como estas texturas são para ser aplicadas a uma superfície, devemos usar sempre `GL_TEXTURE_2D`.

Agora posso, opcionalmente, alterar alguns parâmetros da textura relativos à forma como a mesma deve ser exibida no ecrã e/ou comportar-se. Por exemplo:

```
glTexParameteri(
      GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
      GL_LINEAR_MIPMAP_NEAREST );
```

```
glTexParameteri(
      GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
      GL_LINEAR );
```

As funções `glTexParameteri()` especificam o comportamento da textura. O segundo argumento especifica o atributo a especificar e pode ser (entre outros):

- `GL_TEXTURE_MIN_FILTER` – Filtro a usar quando a textura vai ser reduzida em termos de *pixels*.
- `GL_TEXTURE_MAG_FILTER` – Filtro a usar quando a textura vai ser ampliada em termos de *pixels*.

Por fim, carrego a imagem para essa textura. Isto pode ser feito com:

```
glTexImage2D(
      GL_TEXTURE_2D, 0, GL_RGB,
      igl.width(), igl.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, igl.bits() );
```

ou, se quiser que o OpenGL prepare várias versões "pequenas" *(mipmaps*[*]*)* da imagem que lhe dou, com:

```
gluBuild2DMipmaps(
      GLU_TEXTURE_2D, GLU_RGB,
      igl.width(), igl.height(),
      GLU_RGBA, GLU_UNSIGNED_BYTE, igl.bits() );
```

As constantes `GLU_` são idênticas às `GL_` de mesmo nome. Nestas funções, `GL_RGBA` e `GL_UNSIGNED_BYTE` são o formato que o Qt usa na função `QGLWidget::convertToGLFormat()`. `GL_RGB` (ou a constante `3` nos exemplos NeHe) indica como é que prefiro que o OpenGL guarde a imagem internamente (neste caso, em formato RGB e sem o canal alfa).

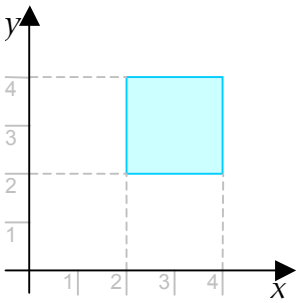
Para mais detalhes sobre estas funções, consulta o manual do OpenGL.

Este processo deve ser repetido para todas as texturas, na inicialização do OpenGL (e.g.: função `View3D::View3D()` do CGMaze). Devemos também activar texturas bidimensionais com:

```
glDisable( GL_TEXTURE_1D );
glEnable ( GL_TEXTURE_2D );
```

[*] *"mip"* do Latim *"multum in parvo"* ou "muito num espaço pequeno". Isto garante uma maior velocidade e qualidade das imagens quando exibidas numa superfície pequena.

Para usar agora esta textura no desenho de um quadrilátero, definido como:

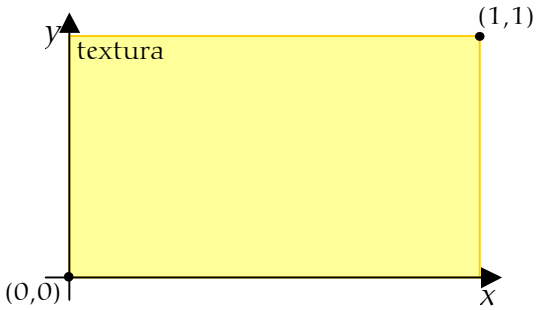


uso o seguinte código:

```
glBindTexture( GL_TEXTURE_2D, texturas[0] );
glBegin( GL_QUADS );
      glTexCoord2i(0,0);  glVertex3i(2, 2, 0);
      glTexCoord2i(1,0);  glVertex3i(4, 2, 0);
      glTexCoord2i(1,1);  glVertex3i(4, 4, 0);
      glTexCoord2i(0,1);  glVertex3i(2, 4, 0);
glEnd();
```

A função `glTexCoord2i()` deve ser usada antes de especificar cada ponto do quadrilátero e ela indica, para o ponto que vai ser desenhado, qual o canto da textura que corresponde a esse ponto.

As coordenadas da textura são sempre definidas como sendo:



Assim, ao ponto (vértice) (2,2) do quadrilátero, corresponde o canto (0,0) da textura. Então uso `glTexCoord2i(0,0)` antes de `glVertex3i(2,2,0)`.

Exercícios

Compila e corre o exercício NeHe ch3.

- Qual é a transparência da cor de fundo usada nesse exercício?
- Repara como o OpenGL cria transições de cores entre pontos que tenham cores diferentes (o triângulo). Altera então o exercício para teres o triângulo roxo e o quadrado com a metade superior azul-mar e a inferior amarelo-alaranjado.

Compila e corre o exercício NeHe ch6.

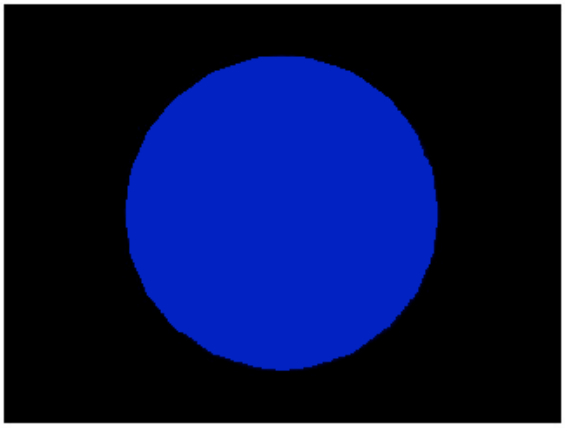
- Altera o cubo gerado para que passe a ter o aspecto de um caixote de madeira (dica: procura uma imagem adequada na mesma pasta onde está a actual).
- Faz com que o código gere uma textura simples encarnada se não encontrar o ficheiro da textura. Testa-o.
- Faz com que duas das faces contíguas do caixote passem a ter o aspecto de uma paisagem. O "chão" de ambas as faces deve estar colado um com o outro.

Aula 08

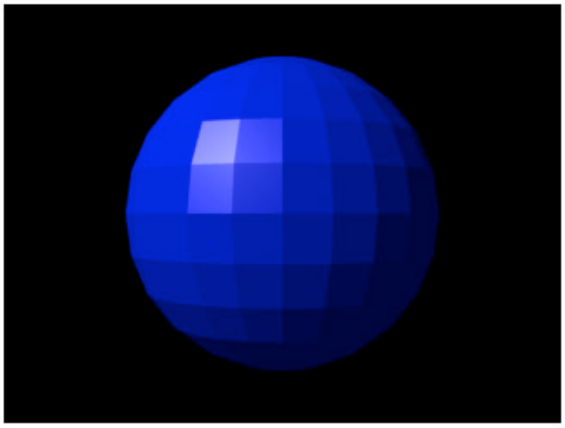
Iluminação (luz ambiente e focos, luz cromática), sombreamento.

Iluminação

Vimos na aula anterior que a luz que incide sobre uma superfície é o que lhe dá a cor. E uma imagem onde não incida nenhuma luz está preta. Em OpenGL podemos simular essa iluminação: isto dará mais realismo às superfícies.



Esfera com todas as superfícies "pintadas" da mesma cor



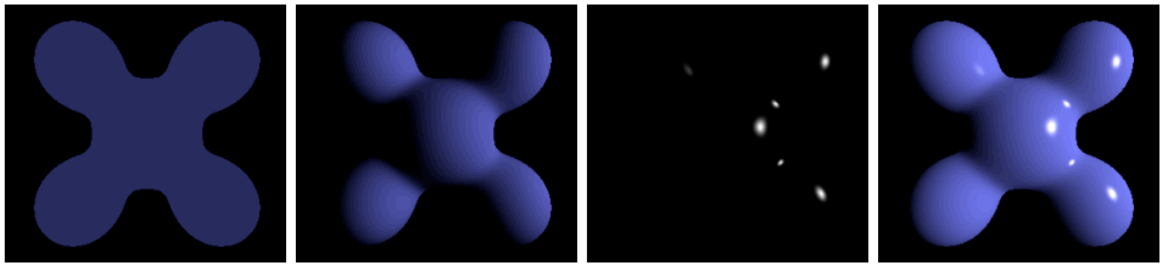
Esfera com cada superfície "pintada" com uma cor diferente dependendo da orientação para a fonte de iluminação

Existem quatro componentes diferentes de iluminação:

- Luz ambiente (`GL_AMBIENT`) – Quando esta fonte de luz é reflectida por todos os objectos da cena, ela ilumina de novo cada objecto, com menos intensidade, mas vinda (para todos os efeitos) "de todos os lados". Esta componente dá o mesmo tipo de iluminação independentemente da direcção em que esteja virada cada face do objecto.
- Luz difusa (`GL_DIFFUSE`) – Componente da iluminação que é reflectida directamente pelo objecto, e portanto a quantidade de luz reflectida por cada face do objecto para o observador depende do ângulo entre o observador, a face e a fonte de iluminação (não depende da distância à fonte, só dos ângulos). Algumas faces serão mais claras e outras mais escuras devido a essas diferenças de ângulo. Este é a componente que ajuda a identificar a forma de um objecto.

- Luz especular (`GL_SPECULAR`) – Componente que representa o reflexo directo da fonte de iluminação em si (habitual em fontes artificiais a iluminar objectos brilhantes). Depende do ângulo entre o observador, a face e a fonte de iluminação, mas também da distância à fonte. Este é o tipo de luz que ajuda a identificar a textura de uma superfície.
- Emissão de luz – quando o próprio objecto emite luz, em todas as direcções.

Estas componentes estão presentes em cada fonte de iluminação, e esta é a base do sistema de iluminação (ou sombreamento) Blinn-Phong usado pelo OpenGL[*]:



Ambiente + Difusa + Especular = Phong

No exemplo acima, as componentes ambiente e difusa da luz são azuis e a componente especular é branca.

Em OpenGL activo os cálculos de iluminação com:

```
glEnable( GL_LIGHTING );
```

Existe uma quantidade finita de fontes de iluminação *(lights)* que podemos criar na nossa cena. O mínimo de fontes de iluminação suportadas é de oito, mas o valor difere de GPU para GPU. A quantidade suportada na placa gráfica onde está a ser compilado o programa é dado por `GL_MAX_LIGHTS`.

As funções para configurar as fontes de iluminação são as seguintes:

```
glLightf ( fonte, parametro, float valor     );
glLightfv( fonte, parametro, float valores[] );
glLighti ( fonte, parametro, int   valor     );
glLightiv( fonte, parametro, int   valores[] );
```

Aqui, `fonte` identifica qual a fonte de iluminação estamos a configurar. Pode assumir o valor `GL_LIGHT0` para a 1ª fonte, `GL_LIGHT1` para a 2ª, etc.. `parametro` identifica qual o parâmetro que estamos a configurar e `valor` ou `valores` será o valor a atribuir a esse parâmetro (se o parâmetro precisa de mais de um valor para ser configurado, precisamos de usar as funções que recebem vectores).

`parametro` pode assumir um dos seguintes valores (entre outros):

- `GL_AMBIENT` – controla a propriedade ambiente desta fonte de iluminação. É definida como um vector de 4 valores `float` RGBA, onde 1.0 representa

[*] Imagem retirada da Wikipedia; autor "Rainwarrior".

a intensidade máxima e -1.0 a intensidade mínima. Esta é a cor da luz após ser reflectida por todos os objectos da cena.

- `GL_DIFFUSE` – controla a propriedade difusa desta fonte de iluminação. É definida como um vector de 4 valores `float` RGBA, onde 1.0 representa a intensidade máxima e -1.0 a intensidade mínima. Esta é a cor da luz quando é reflectida directamente por cada objectos da cena.
- `GL_SPECULAR` – controla a propriedade especular desta fonte de iluminação. É definida como um vector de 4 valores `float` RGBA, onde 1.0 representa a intensidade máxima e -1.0 a intensidade mínima. Esta é a cor da luz quando observada directamente.
- `GL_POSITION` – controla a posição da fonte de iluminação. É definida como um vector de 4 valores `float`, que são as coordenadas *x*, *y*, *z*, *w* da fonte. *w*, como de costume, deve ser 1.0, mas se for 0.0, OpenGL assume que a fonte é direccional (i.e., está infinitamente distante). Esta posição é movida pela matriz `GL_MODELVIEW`, como qualquer outro vértice na cena.

Se quiser ter um foco de iluminação *(spot light)*, `parametro` assume um dos seguintes valores (entre outros):

- `GL_SPOT_DIRECTION` – controla a direcção do foco de iluminação. É definida como um vector de 3 valores `float`, que são as coordenadas *x*, *y*, *z* do vector que indica a direcção da fonte. Esta posição é movida pela matriz `GL_MODELVIEW`, como qualquer outro vértice na cena.
- `GL_SPOT_CUTOFF` – controla a abertura do foco de iluminação. É definida como um valor `float`, que é o ângulo (em graus) em roda do vector direcção do foco, em que este ilumina.
- `GL_SPOT_EXPONENT` – controla a concentração do foco de iluminação. É definida como um valor `float`, quanto mais alto, maior o grau de concentração do foco (independentemente de `GL_SPOT_CUTOFF`).

Tenho de activar cada fonte de iluminação individualmente:

```
glEnable( fonte );
```

Posso ainda especificar a iluminação ambiente para toda a cena com:

```
GLfloat rgba = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv( GL_LIGHT_MODEL_AMBIENT, rgba );
```

Este é o valor de omissão para a iluminação ambiente de toda a cena.

Iluminação: Materiais dos objectos

Cada objecto desenhado em OpenGL simula um objecto do mundo real, esse feito num qualquer tipo de material. Cada material reage de forma diferente às componentes ambiente, difusa e especular de cada fonte de iluminação. Isto representa-se pelas cores do próprio material em cada uma dessas componentes.

Defino isto com:

```
glMaterialfv( face, parametro, float valores[] );
glMaterialiv( face, parametro, int   valores[] );
```

Aqui, `face` identifica qual a face que estamos a configurar. Pode assumir os valores `GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK`. `parametro` identifica qual o parâmetro que estamos a configurar e `valores` serão os valores a atribuir a esse parâmetro.

`parametro` pode assumir um dos seguintes valores (entre outros):

- `GL_AMBIENT`, `GL_DIFFUSE`, `GL_AMBIENT_AND_DIFFUSE` e `GL_SPECULAR` – controla as respectivas propriedades deste material. É definida como um vector de 4 valores `float` RGBA, onde 1.0 representa a intensidade máxima e -1.0 a intensidade mínima. O componente alfa é ignorado em todas as propriedades excepto na respeitante à luz difusa.
- `GL_EMISSION` – controla a propriedade de emissão de luz deste material. É definida como um vector de 4 valores `float` RGBA, onde 1.0 representa a intensidade máxima e -1.0 a intensidade mínima.
- `GL_SHININESS` – controla o expoente especular deste material. É definida como um vector de 1 valor `float`.

Quando usamos `glMaterial()`, deixamos de especificar as cores dos vértices que vamos desenhar com `glColor()`. Isso significa que quando começamos a desenvolver o nosso motor gráfico sem usar iluminação (usando `glColor()`), temos de alterar o código que já tínhamos até esse momento quando introduzimos iluminação.

Existe no entanto uma funcionalidade OpenGL para evitar esta reescrita. É muito comum especificar apenas as propriedades `GL_AMBIENT_AND_DIFFUSE` do material com a mesma cor do objecto em si. Então podemos dizer ao OpenGL para usar a chamada `glColor()` para especificar essa cor.

Fazemos isto com:

```
glEnable( GL_COLOR_MATERIAL );
glColorMaterial( face, parametro );
```

Depois destas chamadas, cada chamada a `glColor()` funciona da mesma forma que uma chamada a `glMaterial()` com a mesma cor.

Por fim, se usares texturas, assegura-te que tens esta linha no código:

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
           GL_MODULATE );
```

ou que a chamada `glTexEnv()` não existe já que `GL_MODULATE` é o valor por omissão. Se quiseres que os focos especulares sejam desenhados, usa ainda isto:

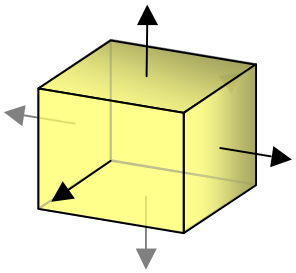
```
glLightModel( GL_LIGHT_MODEL_COLOR_CONTROL,
              GL_SEPARATE_SPECULAR_COLOR );
```

Só funciona em OpenGL 1.2.

Iluminação: Normais

O ângulo que cada fonte de iluminação faz com cada superfície determina a quantidade de luz que ela reflecte e/ou a sua cor. Esse ângulo não é calculado automaticamente. Eu devo indicar um vector (chamado de "vector normal") para cada vértice, que é perpendicular à face e "aponta para fora" do objecto. Se todos os vértices do objecto tiverem a mesma normal, só a precisamos de especificar ao início do `glBegin()`.

Por exemplo, o cubo da aula anterior seria definido com os seguintes vectores:



Nota no entanto que o OpenGL assume três coisas em relação aos vectores normais indicados:

1. Têm origem na origem de coordenadas,
2. Estão de facto perpendiculares à superfície,
3. Têm norma ou dimensão de 1.

Se estas três regras não forem perfeitamente cumpridas, a luz reflectida não será realista.

Defino o vector normal de uma superfície usando:

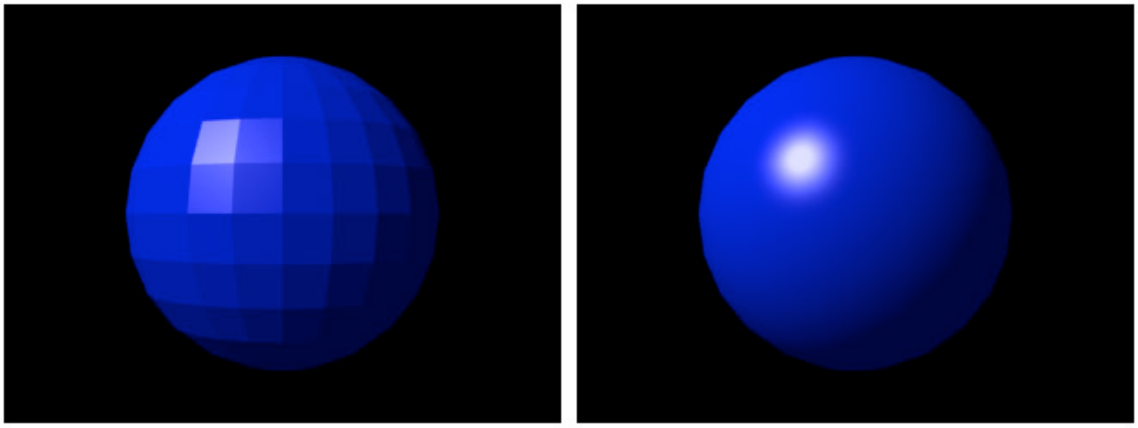
```
glNormal3f( x, y, z );
```

Existem outras variantes da função para aceitar outros tipos de dados, como é habitual em OpenGL.

Como o vector é transformado pela matriz `GL_MODELVIEW`, como qualquer outro vértice na cena, ele pode deixar de ter norma 1. Isto só acontece se o redimensionar, mas posso precisar de o fazer. Posso pedir ao OpenGL para automaticamente normalizar os vectores normais com:

```
glEnable( GL_NORMALIZE );
```

A vantagem de cada normal ser associada aos vértices e não à superfície em si, é que quando estamos a usar polígonos para nos aproximarmos de uma superfície curva (vê exemplo ao início da aula 06, e abaixo), podemos especificar cada normal de cada vértice com o valor correcto para a perpendicular em relação à superfície curva original nesse ponto! Dessa forma, o OpenGL calcula valores reflectivos independentes e preenche a superfície com um gradiente (transição) de cores que irá dar uma maior ilusão de superfície curva.



Uma normal por cada superfície | Uma normal por cada ponto + Phong

Chama-se a este método *Gouraud shading* e faz parte integrante do *Blinn-Phong shading* usado pelo OpenGL.

Exercícios

Compila e corre o exercício NeHe ch7. Quando correres este programa podes usar as setas para acelerar o cubo nos seus eixos, PgUp e PgDown para ampliar/reduzir, L para activar/desactivar a iluminação *(lighting* – mais sobre isto em aula posterior), B para activar/desactivar a transparência *(blending)* e F para iterar por 3 filtros (para qualidade da imagem – mais sobre isto em aula posterior).

- Muda a iluminação difusa para passar a ser com uma luz encarnada.
- Muda a posição da luz para passar a estar no canto superior esquerdo, no plano dos *z*.

Compila e corre o exercício NeHe ch8. Ele tem disponíveis as mesmas teclas de controlo que o exercício ch7.

- Localiza no código as funções que definem os vectores normais.
- Altera todas as normais para que o OpenGL julgue que as faces de "frente" do cubo sejam as faces de dentro.

Aula 09

Modelos 3D em ficheiros.

Animação de objectos na cena: Física e detecção+gravação de movimento humano (sistemas de aquisição de coordenadas para modelação 3D).

O jogo CGMaze: Modelação da vista interior 3D.

Modelos 3D em ficheiros

Quando passarmos para o próximo nível e precisarmos de ter todo um modelo de um mundo ambiente e objectos de cenário, precisaremos de os ter gravados em ficheiros para serem carregados por código nosso para o sistema OpenGL.

O exercício NeHe ch10 é um exemplo desta técnica.

Existem vários programas de desenho de modelos 3D: 3ds Max, Maya, SoftImage XSI, entre outros. Alguns estão especializados para um certo tipo de "motor 3D" (i.e., código de geração OpenGL), como o UnrealEd*.

O problema é que cada um tem o seu próprio formato de ficheiros, optimizado para os seus motores 3D. Quando escolheres um deles, terás de ter informação do seu formato para o conseguires ler e usar em chamadas OpenGL.

O exercício NeHe ch10 usa um ficheiro com listas de 5 coordenadas para cada vértice de um triângulo. As primeiras 3 coordenadas são as *x*, *y* e *z* do vértice. As duas seguintes são as *x* e *y* da textura, nesse vértice.

Este formato é tão válido como qualquer outro. A questão é quando os leres (e a forma de ler depende do sistema que suporta o OpenGL e não do OpenGL em si), o fazeres para um formato que te simplifica a vida mais tarde fornecer esses dados ao OpenGL.

Repara nas variantes da função `glVertex()` que tens disponíveis:

```
glVertex2s( short  x, short  y );
glVertex2i( int    x, int    y );
glVertex2f( float  x, float  y );
glVertex2d( double x, double y );
glVertex3s( short  x, short  y, short  z );
glVertex3i( int    x, int    y, int    z );
glVertex3f( float  x, float  y, float  z );
glVertex3d( double x, double y, double z );
glVertex2sv( short  v[2] );
```

* Vê http://www.udk.com/.

```
glVertex2iv( int    v[2] );
glVertex2fv( float  v[2] );
glVertex2dv( double v[2] );
glVertex3sv( short  v[3] );
glVertex3iv( int    v[3] );
glVertex3fv( float  v[3] );
glVertex3dv( double v[3] );
```

Destas, as mais interessantes são as que recebem vectores. Quando lemos os dados de vértices de um ficheiro dá imenso jeito colocar os dados em vectores. E nesse caso, poupamos tempo de CPU e GPU se usarmos as funções `glVertex()` vectoriais.

Consulta o manual OpenGL para descobrires outras funções OpenGL com facilidades semelhantes.

Física

O movimento de objectos numa cena deve seguir as regras da Física, para ser realista. As leis de Newton que envolvem forças, acção-reacção, velocidade constante e acelerações, atrito e outros factores devem ser consideradas.

Estes cálculos não fazem parte do OpenGL em si, e normalmente são calculados pelo CPU com programação específica. Algumas placas gráficas da marca NVIDIA têm aceleração específica para Física:

http://www.nvidia.com/object/physx_new.html

Outros fabricantes (NVIDIA e ATI) permitem que o programador use parte do poder de computação do GPU para cálculos arbitrários o que deveria permitir que esses cálculos sejam usados para Física.

Para mais informações, lê:
http://en.wikipedia.org/wiki/Physics_engine

Detecção de movimento humano

Animar uma figura humanóide (i.e., com aparência humana) numa cena 3D normalmente não gera movimentos realistas. Por essa razão, quando se pretendem movimentos realistas, eles são capturados de um actor num ambiente controlado, e passados para a animação do modelo humanóide.

A Lusófona tem um laboratório desses: http://movlab.ulusofona.pt/cms/

Para mais detalhes sobre esta tecnologia, consulta o *site*.

Desenhar a vista interior do labirinto em 3D

Para o projecto final, vais precisar de desenhar a vista interior do labirinto do CGMaze, usando OpenGL 3D. Para fazer isso vais usar parte das técnicas de desenhar o mapa com parte das técnicas da vista interior 2D.

Na inicialização (função `View3D::View3D()`), vais carregar as texturas e inicializar as matrizes de perspectiva. Ela recebe um argumento `Map *map` que deve ser copiado para `this->map` (isto já está no código que te é passado) e um argumento `const QImage textures[VIEW3D_TEXTURES_NUMBER]` que é um vector com as imagens para as texturas, já convertido para formato OpenGL com `QGLWidget::convertToGLFormat()`. `VIEW3D_TEXTURES_NUMBER` é a quantidade de texturas. Cada textura nesse vector identifica-se com uma das constantes começada por `VIEW3D_IX_TEXTURE_` que está no ficheiro `student_view3d.h`.

Na função `View3D::resize()` vais ter a função OpenGL `glViewport()` do costume. Mas desta vez, como a função OpenGL `gluPerspective()` também precisa das dimensões reais do *viewport*, ela também vai ser chamada aqui para ajustar a relação de aspecto da perspectiva.

A função `View3D::paint(x, y, compass_direction)` é a mais complexa, e aquela onde se centra a avaliação. Repara como desta vez, e ao invés da visão interior em 2D, esta função recebe todos os argumentos de tipo `float`. Isto é porque o restante código do professor cria uma animação suave de movimento fazendo chamadas sucessivas a `paint()` com valores fraccionários de coordenadas do mapa ou da bússola.

`x` e `y` continuam a referir-se a coordenadas do mapa. (1,1) é a coordenada inicial do jogador. Conforme ele se mover 1 posição para Norte, `paint()` irá ser chamada com valores de `y` com (por exemplo) `1.1`, `1.2`, `1.3`, ..., `1.9` e `2.0`. Conforme ele rodar de Norte para Este (direita), `compass_direction` vai ter valores de `2.1`, `2.2`, ..., `2.9` e `3.0` (vê `compass.h`).

Vais ter então de resolver 2 problemas:

1. Mover e rodar a cena para "fingir" que o jogador avançou dentro do labirinto. Isto se quiseres desenhar cada parede do labirinto nas mesmas coordenadas em que ela está no mapa (como fizeste quando desenhaste o mapa).
2. Encontrar uma forma de desenhar apenas parte do labirinto que está visível, de forma a poderes suportar tamanhos muito grandes de labirintos.

O primeiro problema é razoavelmente fácil de resolver se usares a função OpenGL `gluLookAt()` de que falámos na aula 03.

O segundo problema também pode começar por ser abordado de uma forma simples. Podes desenhar cegamente todo o mapa em 3D. O `gluLookAt()` irá garantir que só será desenhada a parte que está em frente ao jogador.

Fazes isso varrendo todo o mapa da seguinte forma:

```
Cell c;
int mx, my;

for( my = 0;  my < map->getHeight();  my++ )
     for( mx = 0;  mx < map->getWidth();  mx++ )
          {
          c = map->getCell( mx, my );
          // agora uso
          //     if(c.isWallOrDoor())...else...
          // para decidir entre desenhar uma
          // parede ou corredor, respectivamente
          }
```

Depois podes tentar resolver da mesma forma que foi feito na aula 05 para a vista interior em 2D. Fazes o mesmo ciclo que avança "em frente" até encontrar uma parede de frente ou chegar a uma profundidade máxima. Mas vais reparar que isso não resolve o problema dos corredores laterais conforme o jogador se vira. Isso fica para tu resolveres.

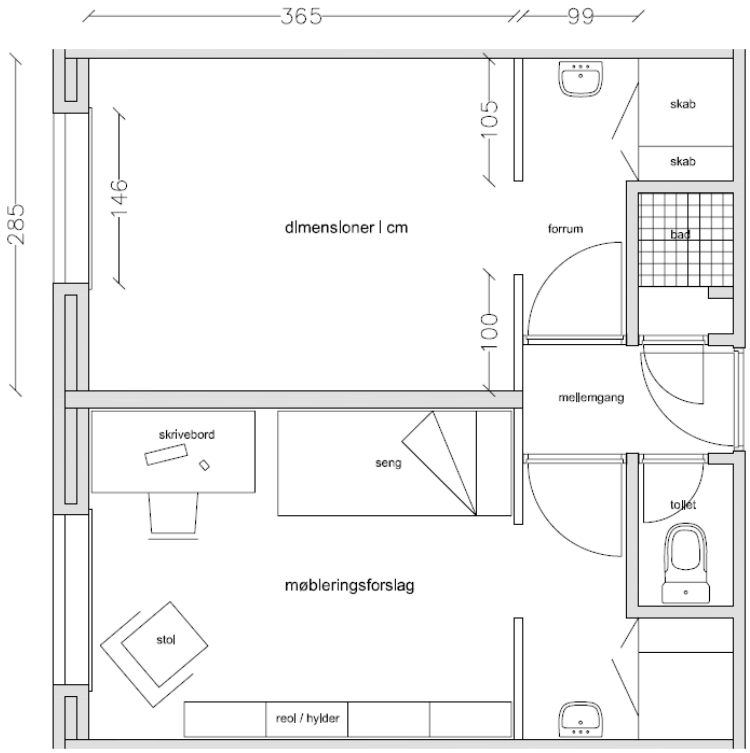
Daí em diante usa a tua criatividade para resolver os restantes problemas do enunciado!

Exercícios

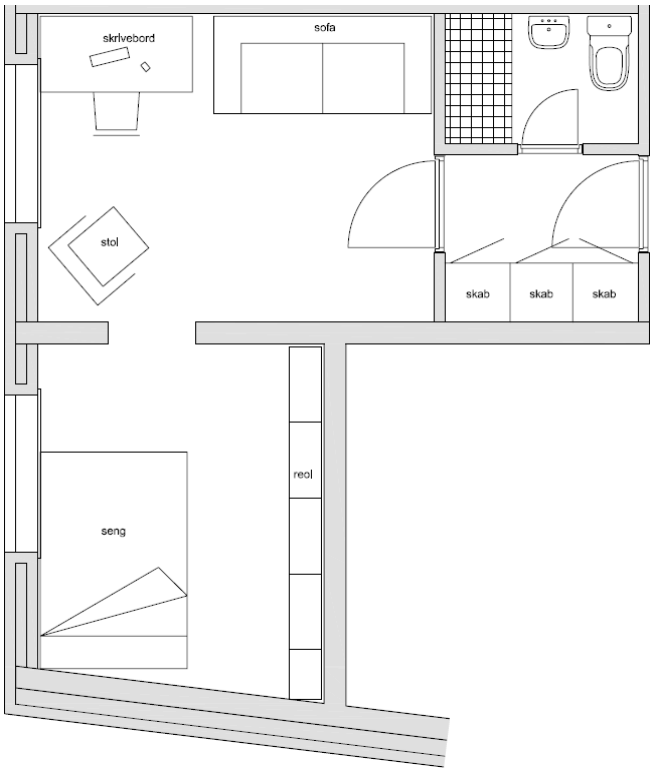
Compila e corre o exercício NeHe ch10.

Altera o ficheiro do mapa (`world.txt`) para poderes navegar numa representação aproximada de um dos quartos do Egmont H. Petersen's Kollegium em Copenhaga, Dinamarca.

Escolhe fazer o mapa de dois quartos ligados por um corredor:



Ou do quarto duplo:



O essencial são as paredes, chão e tecto.

Se tiveres tempo, não te esqueças de aplicar texturas e fazeres alguma mobília que tornem a cena realista!

Aula 10

Tópicos avançados: *Ray tracing*, fractais, sistemas de partículas.

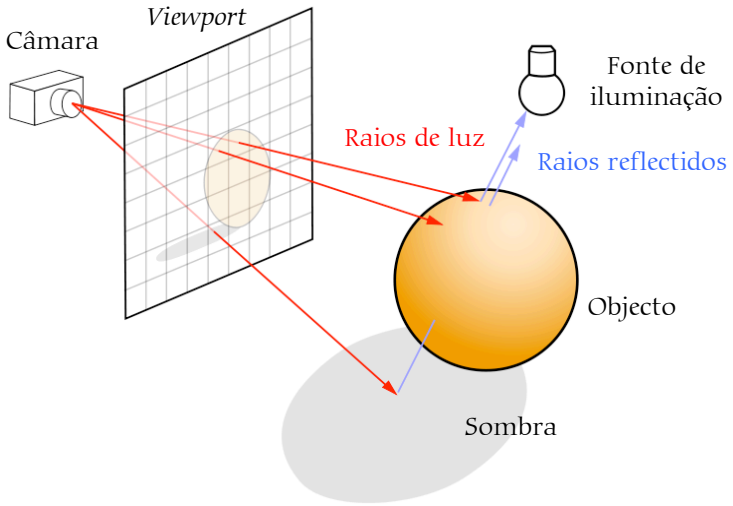
Ray Tracing

OpenGL e outras interfaces de 3D estão preparadas para as placas gráficas desenharem as superfícies (polígonos) que lhes são apresentadas. Embora iluminação e técnicas de sombreamento adicionem realismo às imagens, nunca são completamente fiéis à realidade (especialmente em reflexões e refracções).

Ray Tracing consegue altíssimos graus de realismo:

Esta técnica traça os raios de luz de cada *pixel* da imagem gerada. Este rastreio é feito de forma inversa: o raio que chega a cada *pixel* de uma câmara fotográfica ou de filmar vem da luz reflectida pela cena. Em *Ray Tracing* o raio é rastreado do *pixel* em direcção à cena, até se encontrar uma fonte de iluminação. Em cada superfície, os dados de ângulo com a superfície, cor, tipo de material, etc. são

levados em conta para determinar a nova direcção (ou direcções) do feixe de luz e (no fim) a cor desse *pixel*[*].



A cena continua a ser descrita em termos de polígonos, tipos de materiais e fontes de iluminação. Só o método de a rasterizar para 2D é que muda.

A grande desvantagem deste método é a sua lentidão.

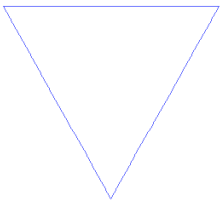
Fractais

Fractais são figuras geométricas que podem ser reduzidas a partes mais pequenas, cada uma das quais é uma cópia reduzida (de pelo menos parte) do todo.

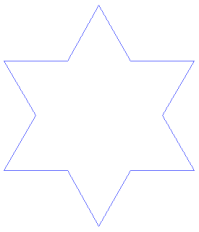
O melhor é explicar com exemplos. O floco de neve Koch começa com um triângulo equilátero. Em cada iteração posterior irá então substituir o 1/3 intermédio de cada face com um triângulo equilátero.

O resultado (nas primeiras 4 iterações) é o seguinte:

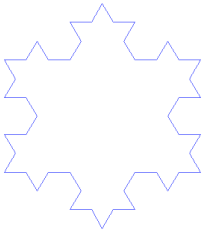
1



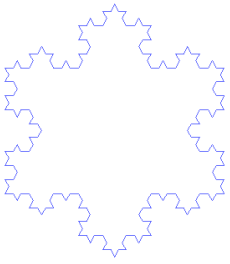
2



3



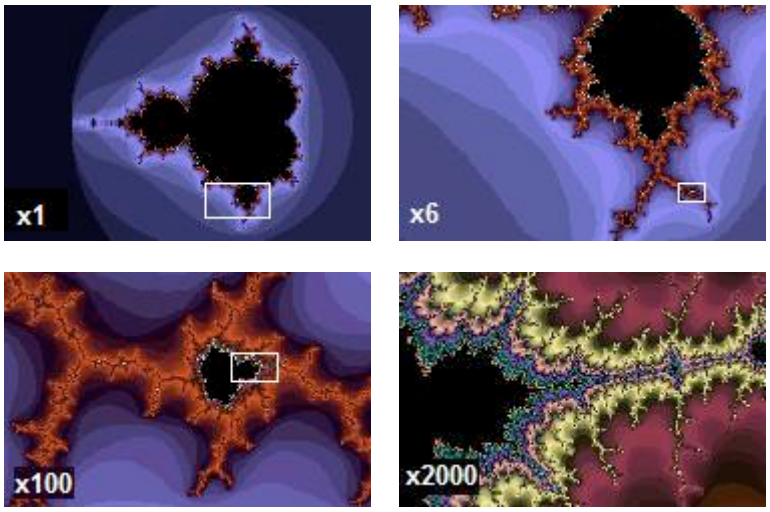
4



[*] Imagem retirada da Wikipedia; autor "Henrik".

Este processo é repetido até cada alteração à imagem seja menos do que um *pixel*. Isto significa que podemos continuar a fazer iterações se o utilizador desejar ampliar a imagem, revelando infinitamente mais detalhes sem perder resolução.

Por exemplo, o conhecido fractal Mandelbrot pode ser repetidamente ampliado:



Como se pode ver neste exemplo, quando amplio o fractal (numa aplicação especial), o fractal é redesenhado com total detalhe para a ampliação pedida, sem perder resolução.

Claro que isto apenas é possível quando a ampliação é executada em software específico de desenho de fractais: se eu colocar um fractal numa imagem JPG ou PNG (ou outra), ela quando ampliada sofrerá de todos os defeitos habituais de uma ampliação digital (pixelização).

Os fractais conseguem boas aproximações de muitas formas da natureza: nuvens, flocos de neve, cristais, cordilheiras montanhosas, relâmpagos, redes de rios, couves-flor e brócolos, redes de vasos sanguíneos e pulmonares, e linhas costeiras.


Brócolo romanesco[*]


Perturbação causada por alta tensão num bloco de acrílico[†]

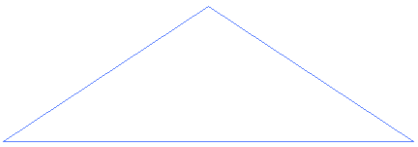
[*] Imagem retirada da Wikipedia; autor "AVM".

[†] Imagem retirada da Wikipedia; autor "Tttrung".

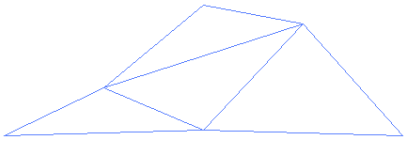
Por exemplo, para criar uma montanha posso começar com um triângulo 3D. Encontro os pontos centrais de cada aresta (lado) e faço um novo triângulo intermédio, cortando efectivamente o triângulo original em 4 novos triângulos. Os vértices do novo triângulo são elevados ou baixados aleatoriamente dentro de uma gama especificada. Em cada nova iteração essa gama é reduzida a metade.

O resultado é o seguinte (em 5 iterações):

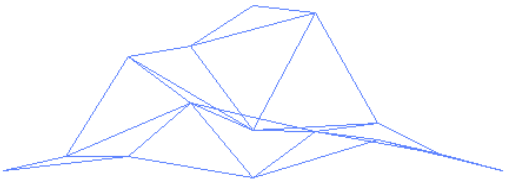
1



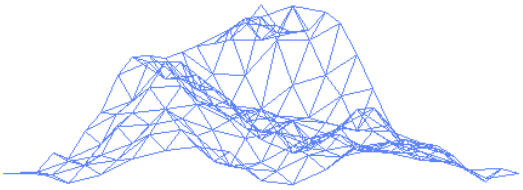
2



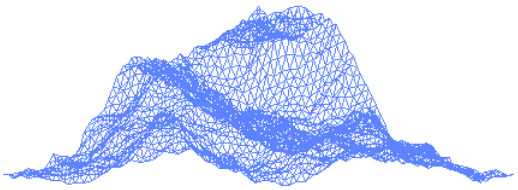
3



4



5

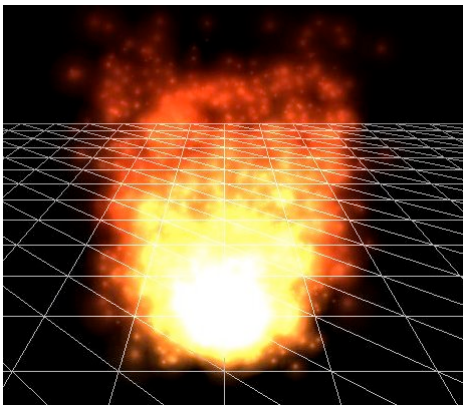


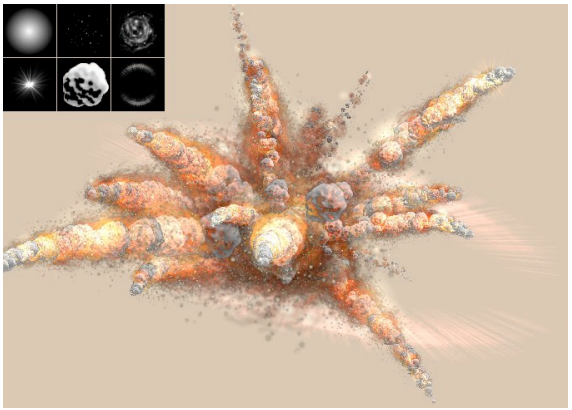
O cálculo de figuras fractais é lento quando comparado com outros métodos de exibição de imagens (JPG, PNG, etc.). No entanto este método tem as suas vantagens e é usado com frequência na geração de terreno e detalhes em cenários tridimensionais e outros fenómenos naturais (plantas, chamas, etc.).

Sistemas de partículas

Sistemas de partículas são simulações de Física (ver aula anterior) em que o objecto ou fenómeno simulado é composto de múltiplos componentes a que chamamos de "partículas".

Exemplos destes sistemas são simulações de chamas e explosões[*]:





Existem alguns motores de sistemas de partículas conhecidos: *The Particle Systems API* (GNU LGPL), a *Dynamic Particle System Framework* para o XNA da Microsoft, *Havok*, *Ageia* (inventora da PhysX da NVIDIA) e *Magic Particles*, entre outros.

[*] Imagem da explosão retirada da Wikipedia; autor "Sameboat".

Projecto

Exercícios e apoio ao projecto.

Exercícios

1. Adiciona texturas especiais para o chão que já foi percorrido pelo jogador. Dica: Vê como no desenho do mapa descobrias quando um pedaço de chão já tinha sido percorrido pelo jogador.
2. Usa (altera) a função `MapCreate::features()` para adicionar aleatoriamente "objectos" a certas partes do chão e paredes. Usa depois esses "objectos" para distinguir entre várias texturas alternativas, de forma a incluir coisas como alçapões fechados, janelas e tochas na parede, etc.. Dica: o objecto não é mais do que um número inteiro associado a uma célula do mapa. Originalmente todos os "objectos" estão a zero, significando "sem objecto". Se `c` for do tipo `Cell`, altero/leio o objecto dentro dele com `c.object`.

Bibliografia

[CGPP]
Foley, van Dam, Feiner, Hughes,
"*Computer Graphics: Principles and Practice*"
(2ª edição), Addison Wesley.
Na Amazon do Reino Unido, em:
http://www.amazon.co.uk/Computer-Graphics-Principles-Practice-International/dp/0321210565/

[QT]
"*Qt*", Nokia.
No *site* da Nokia em:
http://qt.nokia.com/

[NEHE]
"*Neon Helium Productions*", GameDev.
No *site* da GameDev em:
http://nehe.gamedev.net/

[OGL21]
"*OpenGL 2.1 Reference Pages*", Silicon Graphics.
No *site* do OpenGL em:
http://www.opengl.org/sdk/docs/man/xhtml/

[SI]
"*Système International d'Unités*" (Sistema Internacional de Unidades), BIPM.
No *site* do BIPM *(Bureau International des Poids et Mesures)* em:
http://www.bipm.org/en/CGPM/db/11/12/
No *site* da Wikipedia em:
http://pt.wikipedia.org/wiki/Sistema_Internacional_de_Unidades
http://en.wikipedia.org/wiki/International_System_of_Units