

# Computação Gráfica (MIEIC)

## Trabalho Prático 1

### *Geometria básica e transformações geométricas*

## Objetivos

- Instalar, explorar e aprender a utilizar as bibliotecas e exemplos de base para os trabalhos, bem como os procedimentos para a submissão de resultados
- Contactar com as formas mais simples de desenhar formas geométricas básicas utilizando **OpenGL/WebGL**
- Utilizar matrizes de transformação geométrica para manipular/modificar essas formas geométricas
- Utilizar funcionalidades da **WebCGF** para facilitar a definição e aplicação das transformações geométricas

## Introdução

Atualmente é possível gerar gráficos interativos 3D em browsers web, recorrendo à tecnologia **WebGL** e à linguagem **JavaScript**.

Esta forma de desenvolvimento 3D tem as vantagens de não necessitar da instalação de bibliotecas ou a compilação de aplicações, e de poder facilmente disponibilizar as aplicações em diferentes sistemas operativos e dispositivos (incluindo dispositivos móveis). No entanto, para aplicações mais exigentes, é recomendável a utilização de linguagens e bibliotecas mais eficientes, como o **C++** e o **OpenGL**. Porém, dado que a API **WebGL** é baseada em **OpenGL** (mais especificamente **OpenGL ES 2.0**), há uma série de conceitos comuns, pelo que o **WebGL** pode ser visto como uma boa plataforma de entrada nas versões atuais da tecnologia **OpenGL**.

No contexto de **CGRA**, iremos então recorrer ao **WebGL** e a **JavaScript** para criar pequenas aplicações gráficas que ilustrem os conceitos básicos de Computação Gráfica, e permitam a experimentação prática dos mesmos, e que podem ser corridas em *web browsers* recentes (**que suportem WebGL**).

Para apoiar o desenvolvimento, a biblioteca **WebCGF (Web Computer Graphics @ FEUP)** foi desenvolvida pelos docentes da unidade curricular e por alguns estudantes, especificamente para as aulas de **CGRA** e **LAIG**, com o objetivo de abstrair alguma da complexidade de inicialização e criação de funcionalidades de suporte, permitindo a focalização nas componentes relevantes aos conceitos de Computação Gráfica a explorar.

# Preparação do ambiente de desenvolvimento

Uma parte importante deste primeiro trabalho prático é a preparação do ambiente de desenvolvimento. Deve para isso garantir que tem configuradas as principais componentes necessárias, descritas abaixo, e garantir que consegue abrir no browser uma aplicação de exemplo.

## Componentes necessárias

- **Web Browser com suporte WebGL:**
  - A aplicação será efetivamente executada através do browser. Uma lista atualizada dos browsers que suportam WebGL pode ser encontrada em <http://caniuse.com/#feat=webgl> . Os principais browsers para desktop (Google Chrome, Mozilla Firefox, Internet Explorer, Safari) têm atualmente suporte, bem como alguns browsers de dispositivos móveis, como o Google Chrome for Android, e o iOS Safari (embora nalguns casos nem todos os dispositivos que podem correr esses browsers tenham capacidades gráficas para executar as aplicações WebGL).
- **A estrutura de base do projeto:**
  - Inclui a WebCGF e as bibliotecas associadas, bem como a estrutura de pastas de base para o projeto e o ficheiro HTML que serve de base/ponto de entrada da aplicação.
  - Um ficheiro .zip com todos os ficheiros necessários, incluindo o código de base para este trabalho prático está disponibilizado no Moodle.
  - Esta estrutura deve estar numa pasta disponibilizada por um servidor web/HTTP (ver ponto seguinte)
- **Servidor HTTP:**
  - Sendo as aplicações acessíveis via browser, e dadas as restrições de segurança dos mesmos que impedem o acesso a scripts através de ficheiros no disco local, é necessário que as aplicações sejam disponibilizadas através de um servidor web HTTP. No contexto de CGRA, não haverá a geração dinâmica de páginas, pelo que qualquer servidor HTTP que disponibilize conteúdo estático servirá. Existem várias soluções possíveis para este requisito, incluindo:
    - **Usar a área web de estudante da FEUP:** colocar o projeto numa pasta dentro da pasta public\_html da conta do estudante, e acedendo à mesma através do endereço público <http://paginas.fe.up.pt/~eiXXXX/mytest> . Neste caso, ficará por omissão acessível a todos (o que pode ser contornado, p.ex. com um ficheiro de controlo de acesso .htaccess). Tem também a desvantagem de implicar a edição/atualização dos ficheiros no servidor da FEUP, e obrigar a uma ligação à rede da FEUP para poder editar/carregar a aplicação

- **Usar um servidor web no próprio computador:** Nalguns casos os estudantes têm já um servidor web (ex: Apache, Node.js) a correr para suportar outros projetos. O mesmo pode ser usado para este efeito, desde que o servidor disponibilize a pasta com a aplicação através de um URL acessível por HTTP. No caso de não terem nenhum servidor, podem correr um mini-servidor usando uma das seguintes opções:
  - **Mongoose Web Server** - <http://cesanta.com/mongoose.shtml> (Windows/Mac): Um mini-servidor web que pode ser colocado na própria pasta do projeto (ou noutra pasta que a inclua)
  - **Python:** Caso o Python esteja instalado, pode ser criado um servidor HTTP simples correndo, na pasta que se pretende partilhar via web, o comando seguinte (dependendo da versão de Python):
    - `python -m SimpleHTTPServer 8080` (para versões 2.x)
    - `python -m http.server 8080` (para versões 3.x).
- **Um editor de texto ou IDE:** O código que compõe as aplicações será escrito em JavaScript, e armazenado em ficheiros de texto. Para a sua edição existem várias alternativas também:
  - O próprio **Google Chrome** disponibiliza nas suas "**Developer Tools**" (**Ctrl-Shift-I**) um debugger de JavaScript, que permite fazer execução passo-a-passo, análise de variáveis, consulta da consola, etc. ao código que está a ser corrido no browser, e permite também mapear os ficheiros acessíveis por HTTP aos ficheiros originais armazenados em disco. Pode por isso ser usado como editor e debugger, e é no momento **a solução recomendada**.
  - Em alternativa, a utilização de um IDE que suporte JavaScript é recomendada, uma vez que pode auxiliar na deteção de erros de sintaxe e na consulta a informação relacionada com JavaScript. O Eclipse, por exemplo, tem um pacote dedicado a JavaScript em <https://eclipse.org/downloads/packages/eclipse-ide-javascript-web-developers/indigosr2>
  - No limite, qualquer editor de texto pode servir para editar os ficheiros. No entanto, sugere-se fortemente um editor de texto que suporte a navegação numa árvore de ficheiros, para permitir alternar facilmente entre os diferentes ficheiros que constituirão o projeto (Ex: Brackets, Sublime, Atom...).

## Teste do ambiente de desenvolvimento

Nesta fase deve ter já uma pasta com a aplicação/template, partilhada através de um servidor web. Deve por isso ter também o endereço URL através do qual a pasta é acessível (Evitar pastas com espaços e acentos!). Abra o browser e direcione-o para o URL referido, e ao fim de alguns segundos deve surgir a aplicação de exemplo, podendo manipular o ponto de vista com o rato, usando o botão esquerdo para rodar a cena, o botão direito para a deslocar lateralmente, e carregando no botão central para aproximar/afastar.

# Recursos disponibilizados

## A biblioteca 'WebCGF'

### Estrutura

A biblioteca **WebCGF** (Web Computer Graphics @ FEUP) - tem como classes principais as seguintes:

- **CGFapplication (+)** - Gere as questões genéricas de inicialização da aplicação e bibliotecas de apoio, e interliga os outros componentes
- **CGFscene (\*)** - É responsável pela inicialização, gestão e desenho da cena
- **CGFinterface (\*)** - É usada para construir e gerir a interface com o utilizador; pode aceder ao estado interno da cena para, por exemplo, ativar ou desativar funcionalidades (p.ex. luzes, animações)

A biblioteca contempla também as seguintes classes que representam entidades que podem integrar uma cena (lista não exaustiva):

- **CGFobject (\*)** - Representa um objeto genérico, que deve implementar o método **display()**; os objetos a serem criados devem ser sub-classes de **CGFobject**
- **CGFlight (+)** - Armazena alguma informação associada a uma fonte de luz (poderá ser estendida por sub-classes para implementar características adicionais)
- **CGFcamera (+)** - Armazena a informação associada a uma câmara

Para a correta execução dos trabalhos, espera-se que **venham a estender as classes assinaladas com (\*)**, de forma a implementar as cenas, interface e objetos requeridos em cada um dos trabalhos, tal como exemplificado na secção seguinte.

As classes assinaladas com **(+)** são **classes utilitárias de exemplo, não exaustivas**, a instanciar para facilitar a gestão e armazenamento das entidades associadas (podendo no entanto ser estendidas por sub-classes, se desejarem acrescentar funcionalidades). A biblioteca inclui ainda alguns objetos pré-definidos, como é o caso dos eixos (**CGFaxis**), e mais algumas classes auxiliares, mas que não deverão ser necessárias para os primeiros trabalhos práticos de CGRA.

### Interação de base

Em termos de interação, por omissão é possível manipular a vista utilizando o rato da seguinte forma:

- Botão esquerdo - rodar a cena em torno da origem
- Botão central (roda pressionada) - aproximar/afastar; em alternativa, pode ser utilizado CTRL + Botão esquerdo
- Botão direito - "deslizar" a câmara lateralmente

## Código de base do exercício

O código de base fornecido para o exercício estende as classes referidas na secção anterior de forma a implementar o desenho de uma cena muito simples.

A classe **TPscene** estende **CGFscene**, e implementa os métodos **init()** e **display()**:

- **init()** : Contém o código que é executado uma única vez no início, depois da inicialização da aplicação. É aqui que tipicamente se inicializam variáveis, criam objetos ou são feitos cálculos intensivos cujos resultados podem ser armazenados para posterior consulta.
- **display()** : Contém o código que efetivamente desenha a cena repetidamente. Este método será o foco deste primeiro trabalho.

Leia com atenção os comentários disponíveis no código desses dois métodos, pois fornecem informação importante sobre o seu funcionamento e utilização.


Em particular, o código de exemplo contido no método **display()** está dividido em três secções:


- Inicialização do fundo, câmara e eixos
- Transformações geométricas
- Desenho de primitivas

Este trabalho focar-se-á no desenho de primitivas e a sua organização, a declaração e uso de transformações geométricas, e a combinação de ambas para produzir uma geometria composta.

# Trabalho prático

Ao longo dos pontos seguintes são descritas várias tarefas a realizar. Algumas delas estão anotadas

com o ícone  (captura de imagem). Nestes pontos deverão, capturar uma imagem da aplicação para disco (p.ex. usando Alt-PrtScr em Windows ou Cmd-Shift-3 em Mac OS X para capturar para a clipboard e depois gravar para ficheiro num utilitário de gestão de imagens à escolha). No final de cada aula, devem renomear as imagens para o formato **"CGFImage-tp1-x.y.png"**, em que **x** e **y** correspondem ao ponto e subponto correspondentes à tarefa (p.ex. **"CGFImage-tp1-2.4.png"**).

Nas tarefas assinaladas com o ícone  (código), devem criar um ficheiro **.zip da pasta que contém o vosso código (tipicamente na pasta 'tp1', se tiverem código noutras pastas incluam-no também)**, e nomeá-lo como **"CGFCode-tp1-x.y.zip"**, (com x e y identificando a tarefa tal como descrito acima).

No final (ou ao longo do trabalho), devem submeter os ficheiros via Moodle, através do link disponibilizado para o efeito. Devem incluir também um ficheiro **ident.txt** com a lista de elementos do grupo (nome e número). Bastará apenas um elemento do grupo submeter o trabalho.

## 1. Geometria básica e estruturação

O desenho de objetos em **WebGL** e nas versões modernas de **OpenGL** é tipicamente baseado na definição de um conjunto de triângulos com um conjunto de características associadas.

Esses triângulos são definidos por um conjunto de vértices (e possivelmente algumas características associadas a cada vértice), e a forma como os vértices se ligam para formar os triângulos.

A título de exemplo, considere-se um retângulo de vértices A, B, C e D, constituído por dois triângulos de vértices ABC e DCB (Figura 1, código correspondente no ficheiro **MyObject.js**).

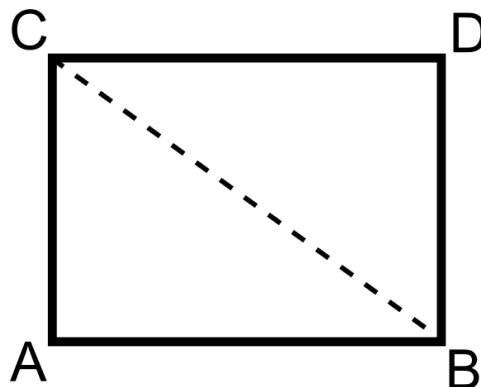


Figura 1: Geometria de exemplo (retângulo).

No código de base fornecido com o trabalho prático é providenciado o código necessário para criar um retângulo, e incluído na cena para que possa ser visível.

Por exemplo, um retângulo com cantos A, B, C e D pode ser definido por dois triângulos unindo os vértices ABC e DCB. Para criarmos essa geometria, temos em primeiro lugar de criar um *array* de vértices com as coordenadas dos quatro cantos.

```
vertices = [  
    xA, yA, zA,  
    xB, yB, zB,  
    xC, yC, zC,  
    xD, yD, zD  
];
```

Em seguida, devemos indicar como estes vértices serão ligados entre si para formar triângulos. Para isso criamos um novo *array* que indique, através de índices referentes à ordem dos vértices, como agrupá-los três a três. Neste caso, sendo os triângulos definidos pela ordem ABC e DCB, teremos:

```
indices = [  
    0, 1, 2,  
    3, 2, 1  
];
```

O uso de índices permite reduzir a quantidade de informação necessária para definir a geometria. Em vez de repetirmos 3 coordenadas na lista de vértices quando o mesmo vértice é usado mais do que uma vez, apenas repetimos o seu índice na lista de índices.

Quanto maior for a geometria e o número de vértices compartilhados (algo bastante comum em modelos 3D compostos por uma malha de triângulos), mais benefício há em usar os índices para representar a conectividade..

Tendo esta informação definida, o desenho efetivo da geometria implica passar a informação assim declarada em **JavaScript** para buffers do **WebGL** (já alocados na memória gráfica), e instruir o mesmo para os desenhar considerando a sua conectividade como sequências de triângulos.

Na **WebCGF**, a complexidade desta fase final do desenho está encapsulada na classe **CGFObject**. Dessa forma, para criar um determinado objeto 3D, podemos simplesmente:

- criar uma sub-classe da **CGFObject**, p.ex. **MyObject**
- implementar o método **initBuffers**, onde
  - declaramos os arrays acima referidos,
  - invocamos a função **initGLBuffers** para a informação ser passada para o **WebGL**
- Na nossa cena:
  - Criar e inicializar uma instância do novo objeto no método *init()* da cena
  - Invocar o método **display()** dessa instância do objeto no método *display()* da cena

## Exercício

1. Modifique o objeto fornecido, de forma a que, além do retângulo já definido, inclua também um triângulo isósceles com uma base de 2 unidades de lado, e uma unidade de altura, assente sobre o retângulo, deixando meia unidade de margem para cada lado (centrado como o telhado de uma casa, ver Figura 2). (📷 1.1)

**Sugestão:** deve alterar o conteúdo da lista de vértices e da lista de índices.

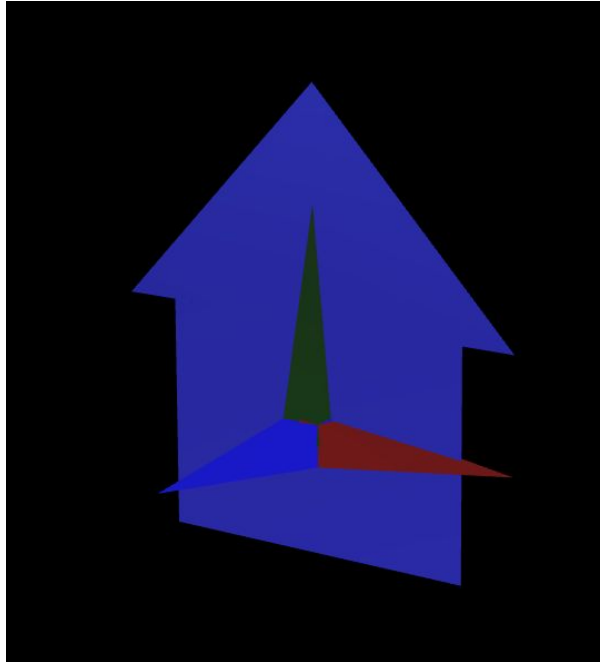


Figura 2: Resultado esperado do Ex. 1.1.



## 2. Matrizes de transformações geométricas

Num sistema de coordenadas 3D, as três transformações geométricas básicas -Translação, Rotação e Escalamento - são representáveis por matrizes quadradas, com 4 linhas e 4 colunas. A concatenação de um conjunto de transformações geométricas obtém-se pela multiplicação das matrizes respetivas. Em **OpenGL/WebGL**, o formato das matrizes de transformações geométricas corresponde ao transposto das matrizes definidas; assim, ao pretender-se uma matriz com o conteúdo seguinte:

```
| 0 1 2 3 |  
| 4 5 6 7 |  
| 8 9 A B |  
| C D E F |
```

deve fazer-se, em **OpenGL/WebGL**:

```
m =[  
    0, 4, 8, C,  
    1, 5, 9, D,  
    2, 6, A, E,  
    3, 7, B, F  
];
```

O método **init()** contém uma secção relativa a transformações geométricas. Nesta secção encontra-se a definição de três matrizes de transformação geométrica (atenção aos comentários que são feitos).

O método **display()** contém outra secção relativa a transformações geométricas com um bloco de concatenação/multiplicação das matrizes anteriores (em comentário, não está a ser utilizada), onde encontra comandos do tipo **this.multMatrix(...)**, que usam o método **multMatrix** da **CGFscene** para acumular essas transformações à perspetiva da câmara, de forma a que os objetos sejam transformados relativamente à mesma.

### Exercício

1. Descomente, a cada vez, uma e uma só das operações de multiplicação de matrizes correspondentes às diferentes transformações, e execute o programa
2. Descomente a aplicação da Translação e da Rotação, mantendo o escalamento em comentário, e execute o programa
3. Troque de ordem as duas transformações anteriores e execute o programa;
4. Coloque novamente todas as linhas na sua ordem e estado inicial, e repita os pontos anteriores,

mas com Translação e Escalamento (  2.4) .

### 3. Funções WebCGF para transformações geométricas

A **WebCGF** fornece na sua classe **CGFscene** um conjunto de instruções que permitem manipular transformações geométricas e aplicá-las à perspectiva da câmara, baseadas na biblioteca **gl-matrix.js**; com elas não é necessário declarar as matrizes nem converter ângulos de graus para radianos. São elas:

- **CGFscene.translate(x, y, z)**: Gera uma matriz de translação e aplica-a;
- **CGFscene.rotate(ang, x, y, z)**: Gera uma matriz de rotação de *ang* radianos à volta do eixo (x, y, z) e aplica-a
- **CGFscene.scale(x, y, z)**: Gera uma matriz de escalamento nas três direções e aplica-a; Nota: nenhum dos componentes deve ser zero, caso contrário a geometria será reduzida a algo planar, com efeitos indefinidos.

Antes da execução das alíneas seguintes, comente toda a secção de transformações geométricas, incluindo as instruções de concatenação (quando visualizado, o objeto constituído pelas duas primitivas deve ter regressado à sua posição original).

#### Exercício

1. Substitua cada uma das três multiplicações de matrizes pela invocação do método da **CGFscene** correspondente à transformação geométrica pretendida, p.ex. **this.translate(...)**;
2. Repita os pontos da secção anterior, descomentando e trocando a ordem das instruções recém-introduzidas. No final deste ponto deve ter o escalamento e a translação ativas.
3. Copie a linha que desenha o objeto para antes da aplicação das transformações. Ao executar o programa deve ter duas cópias do objeto, uma na posição original, e outra na posição e escala resultante das transformações.
4. Aplique agora uma operação de translação de 5 unidades em YY antes da cópia que criou (ou seja, deverá ter a sequência translação, objeto, escalamento, translação, objeto). Note que, ao executar o programa, ambos os objetos foram afetados pela translação que foi adicionada. (



3.4)

5. Recorrendo às instruções **CGFscene.pushMatrix()** e **CGFscene.popMatrix()**, garanta que o segundo objeto fica na mesma posição em que estava no final do ponto 3 (ou seja, apenas

afetado pelo escalamento e translação iniciais). (  3.5) (  3.5).

## 4. Geometria composta - Cubo Unitário

Até agora, foram apenas consideradas superfícies coplanares. Neste exercício pretende-se a criação de um cubo unitário, ou seja, um **cubo centrado na origem e de aresta unitária**, ou seja, com coordenadas entre  $(-0.5, -0.5, -0.5)$  e  $(0.5, 0.5, 0.5)$ .

Vão ser exploradas duas formas de o fazer: construindo uma única malha de triângulos, tal como em exercícios anteriores, ou construindo um objeto composto, em que uma ou mais malhas de triângulos são usadas num mesmo objeto (possivelmente sofrendo diferentes transformações).

Comece por fazer uma cópia da pasta do projeto para seu arquivo, e em seguida apague da função **display()** todo o código anterior relacionado com as transformações geométricas e os dois objetos, de forma a ter o método **display()** apenas a desenhar os eixos (ou seja, remova todo o código entre o desenho dos eixos, e o fim do método **display()**).



### Exercício

1. Crie um ficheiro **MyUnitCube.js** e defina nesse ficheiro a classe **MyUnitCube** como subclasse da **CGFObject** (deve usar uma cópia do código do **MyObject** como ponto de partida). Essa classe deve definir na função **initBuffers** os 8 vértices do cubo, e a conectividade entre eles de forma a formar os triângulos que constituem as faces quadradas do cubo. Recomenda-se que sejam inseridos comentários identificando os vértices e as faces que estão a ser definidas.
2. Deve acrescentar no ficheiro **main.js** a inclusão do novo ficheiro **MyUnitCube.js**, na linha onde são incluídos os outros ficheiros do projeto.
3. Inclua um novo objeto do tipo **MyUnitCube** na função **init** da **TPScene**, e invoque o método **display()** de **MyUnitCube** no método **display()** da **TPScene**. Execute a aplicação. Deve ter um cubo unitário centrado na origem.

( 4.3)

4. Iremos agora criar um novo cubo unitário, recorrendo ao desenho de vários quadrados unitários. Para isso, comece por recuperar o ficheiro **MyObject.js** original do .zip fornecido, e mude o seu nome para **MyQuad.js**, bem como deve substituir o nome dos métodos no seu interior de **MyObject...** para **MyQuad...**
5. De forma análoga à efetuada na alínea 1, crie agora um novo ficheiro **MyUnitCubeQuad.js** e defina nesse ficheiro a classe **MyUnitCubeQuad** como subclasse da **CGFObject**. Não iremos necessitar de buffers no **MyUnitCubeQuad**, já que a geometria de base está definida no **MyQuad**. Retire por isso a invocação de **this.initBuffers()** do construtor, e a própria função **MyUnitCubeQuad.prototype.initBuffers()**.
6. Altere de seguida o construtor do **MyUnitCubeQuad** de forma a adicionar à classe um membro **quad** que é uma nova instância de **MyQuad**, e a inicializá-la (nota: não copie o código diretamente deste documento, poderá incluir caracteres especiais que originem erros na script).

```
this.quad=new MyQuad(this.scene);  
this.quad.initBuffers();
```

7. Deve agora criar a função **MyUnitCubeQuad.prototype.display()**, que será a responsável por desenhar as seis faces do cubo usando o **MyQuad** previamente definido. Nesta função desenhe a face paralela a XY, em  $Z=+0.5$ , de forma semelhante à usada no **MyObject** original, e utilizando uma translação; deve usar **this.scene.translate(...)** para poder aplicar a translação no contexto da cena, e **this.quad.display()** para desenhar a face propriamente dita.
8. Desenhe as outras faces aplicando as transformações necessárias, e invocando o desenho do *quad* as vezes necessárias.
9. De forma análoga ao ponto 2, inclua um **MyUnitCubeQuad** na função *init* da **TPscene**, e invoque o seu método **display()** na função **display** da **TPscene**. No entanto, neste caso **preceda-o de uma translação de 2 unidades em X**, para ficar ao lado do **MyUnitCube** criado anteriormente. Não se esqueça de incluir referência a **MyUnitCubeQuad.js** e a **MyQuad.js** no ficheiro **main.js**
10. Ao executar o programa, deve ter dois cubos unitários, um centrado na origem, e outro duas unidades ao lado (  4.10) (  4.10)

## 5. Geometria composta - Cena

Pretende-se agora implementar o código correspondente a uma cena mais complexa, constituída por uma mesa (tampo e pernas paralelepípedas) e um chão (outro paralelepípedo), tal como apresentado na Figura 3.

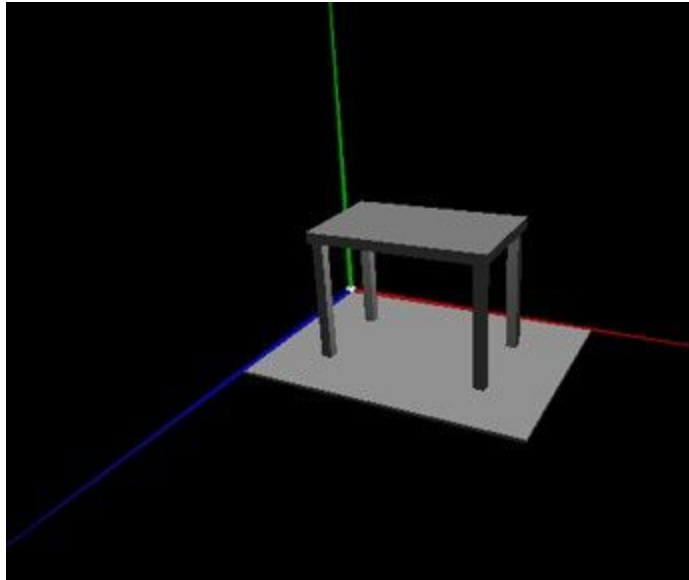




Figura 3: Imagem ilustrativa da cena a desenvolver.



### Exercício

1. Crie agora uma nova classe **myTable**, e substitua na **TPscene** os cubos unitários por **myTable**
2. Acrescente à classe **myTable** uma instância de um objeto **MyUnitCubeQuad**, e acrescente o método **display()** da **myTable** para que aplique transformações e invocações do método **display()** do cubo criado para criar as peças da mesa **assente no plano XZ**, e centrada na origem. A mesa é constituída por paralelepípedos: quatro pernas (**dimensões 0.3\*3.5\*0.3 unidades**) e um tampo (**dimensões 5\*0.3\*3 unidades**).
3. Crie uma classe **myFloor** à semelhança das anteriores, acrescente à **TPscene** uma instância da mesma, e escreva no método **display()** de **myFloor** o código necessário para desenhar o chão, assente no plano XZ, também paralelepípedo, de dimensões (**dimensões 8\*0.1\*6 unidades**).
4. Acrescente no método **display()** da **TPscene** as transformações necessárias para deslocar o chão e a mesa de forma a que duas das arestas do chão coincidam com os eixos X e Z

( 5.4) ( 5.4)

# Checklist

Até ao final do trabalho deverá submeter as seguintes imagens e versões do código via Moodle, **respeitando estritamente a regra dos nomes**, bem como o ficheiro **ident.txt** com a identificação dos membros do grupo:

-  **Imagens (7): 1.1, 2.4, 3.4, 3.5, 4.3, 4.10, 5.4 (nomes do tipo "CGFImage-tp1-x.y.png")**
-  **Código em arquivo zip (3): 3.5, 4.10, 5.4 (nomes do tipo "CGFCode-tp1-x.y.zip")**