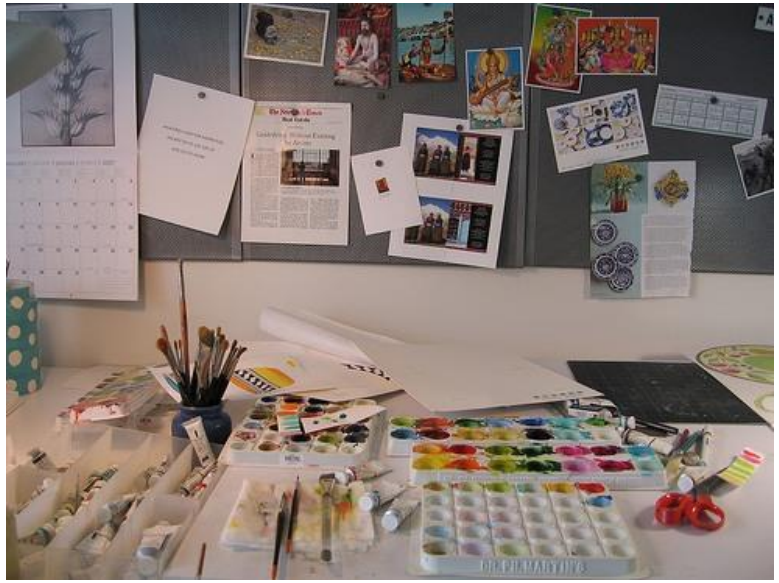
La tubería de fragmentos de OpenGL



http://flickr.com/photos/spychic/

Bibliografía:

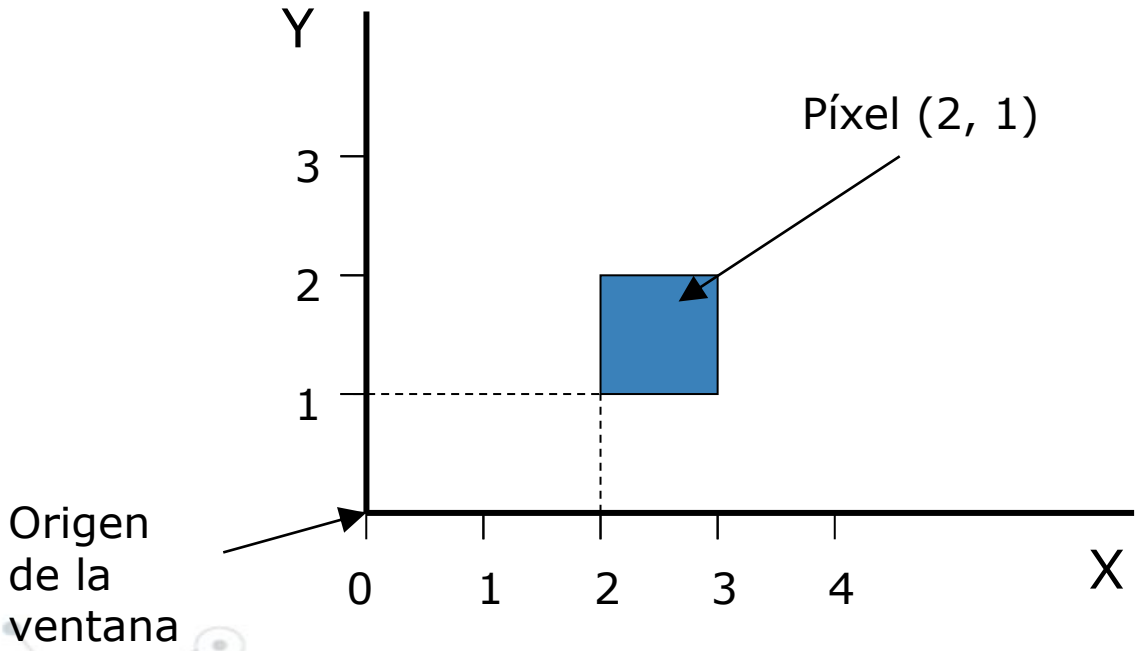
- Superbiblia 7ª ed. Cap. 4, 382-387, Cap. 9.

Introducción

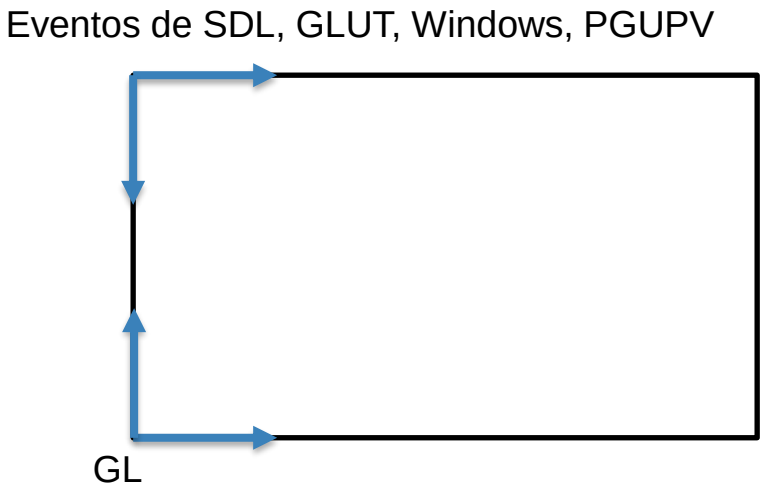
- Un *buffer* es un conjunto rectangular de elementos que contienen información asociada a los píxeles de la ventana (p.e., color, profundidad, etc).
- A partir de OpenGL 3.0, se introdujeron los *Framebuffer objects* (FBO), que facilitan el render fuera de la pantalla y el render a textura

Introducción

- Sistema de coordenadas 2D de los buffers en OpenGL



- El píxel (x, y) ocupa el área cuadrada entre las coordenadas (x, y) y (x+1, y+1)
- ¡Cuidado! En las librerías de apoyo (como GLUT, SDL o PGUPV), las coordenadas de ventana se dan con respecto a la esquina superior izquierda



Introducción

- El *framebuffer* está compuesto por una serie de buffers:
  - *Color* : es el único que puede ver el usuario
    - En el FB por defecto: frontal izquierdo, frontal derecho, trasero izquierdo y trasero derecho.
    - En un FBO definido por el usuario también se pueden definir varios buffer de color
  - *Depth* : almacena la coordenada Z normalizada de cada pixel
  - *Stencil*: almacena un número entero (normalmente un byte) y se suele usar para evitar dibujar en una zona arbitraria de la ventana

El *color buffer*

- Contiene el color de los píxeles que se mostrarán en la ventana, y quizá su alfa.
- Si la implementación de OpenGL soporta visión estereoscópica, entonces habrá un buffer para el ojo izquierdo, y otro para el derecho (si no, sólo está el izquierdo)
- Además, si se usa doble buffer para animación, hay un buffer frontal y otro trasero por cada ojo (si no, sólo hay frontales)
- El programador puede pedir a OpenGL FBO adicionales (no visibles), por ejemplo, para almacenar imágenes que se repiten a menudo o para usarlo como texturas dinámicas

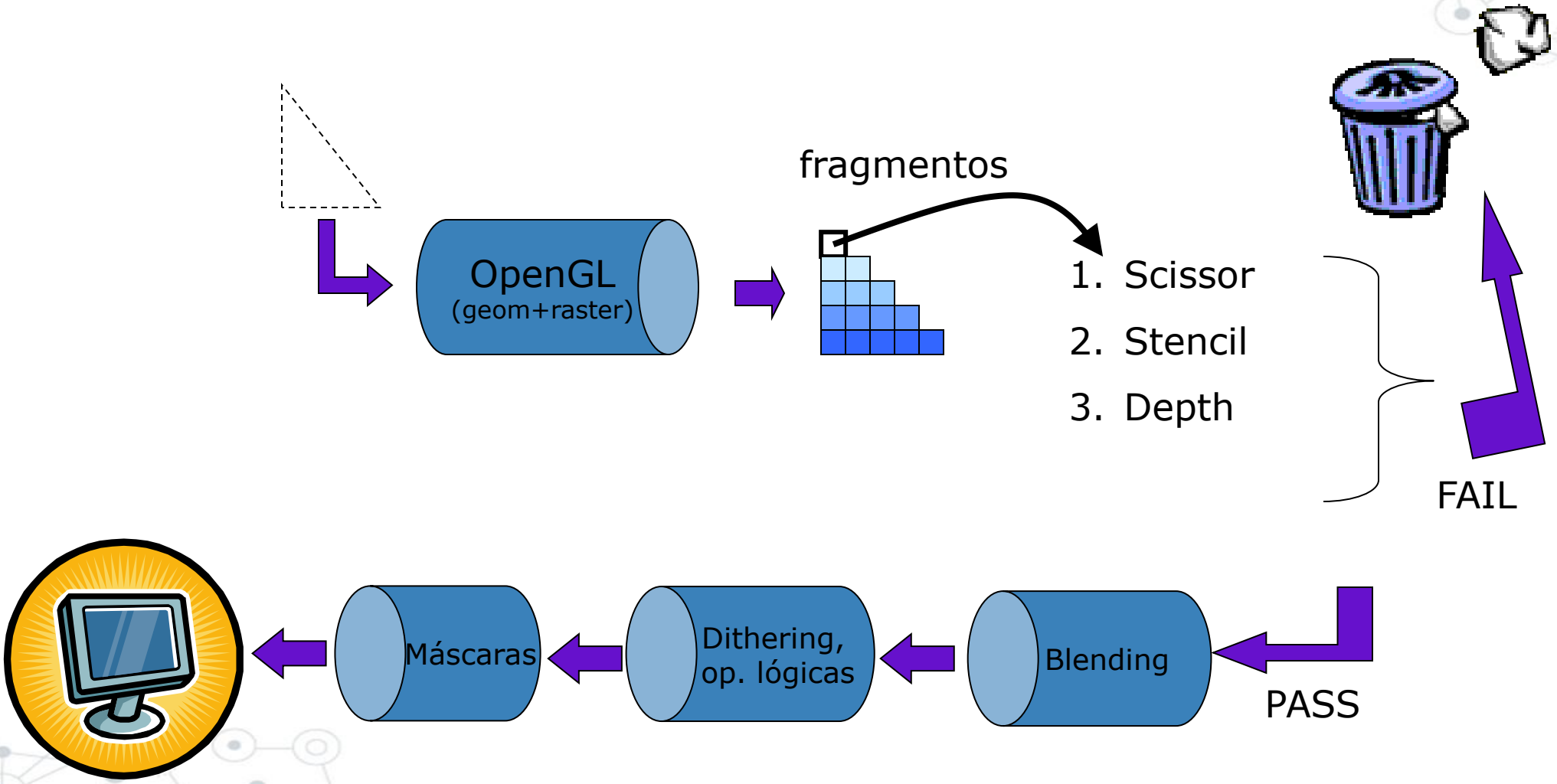
El *color buffer*

- Para averiguar si hay disponible soporte de visualización estereoscópica o doble buffer:
  - **`GLboolean b;`**
  - **`glGetBooleanv(param, &b);`**
  - donde param: GL_STEREO, GL_DOUBLEBUFFER

Borrando buffers

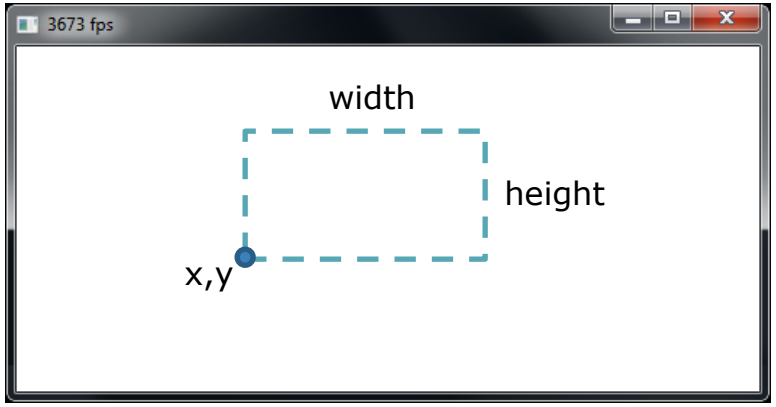
- En OpenGL se borran los buffers en dos pasos:
  - Se establece el valor de borrado para cada buffer
  - Se pasa una lista de los buffers a borrar
- Establecer el valor:
  - `void glClearColor(red, green, blue, alpha); // 0, 0, 0, 0`
  - `void glClearDepth(depth); // 1`
  - `void glClearStencil(s);  // 0`
- Para borrar:
  - `void glClear( GLbitfield mask );`
    - `GL_{COLOR, DEPTH, STENCIL}_BUFFER_BIT`
- A partir de OpenGL 3, se puede borrar un buffer directamente a un color dado en la llamada
  - **`glClearBufferiv, glClearBufferuiv, glClearBufferfv, glClearBufferfi`**

La carrera del fragmento



Scissor test

- Define un rectángulo en la ventana donde se permite el dibujado (no permitido en el resto)
  - **`void glScissor(`*`x, y, width, height`*`);`**
- Para activar este test:
  - **`glEnable(GL_SCISSOR_TEST)`**
- Afecta a **`glClear`**



El *stencil buffer*

- El stencil se suele usar para evitar dibujar en alguna zona de la ventana

http://visualresistance.org

Stencil test

- Necesita un stencil buffer (normalmente *ints* de 8 bits).
  - `myApp.initApp(argc, argv, PGUPV::DOUBLE_BUFFER | PGUPV::DEPTH_BUFFER | PGUPV::STENCIL_BUFFER);`
- Compara un valor de referencia con el valor almacenado en la posición correspondiente del buffer
- Funciones:
  - `void glStencilFunc(GLenum func, GLint ref, GLuint mask)`
  - `void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass)`
  - `void glStencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint mask)`
  - `void glStencilOpSeparate(GLenum face, GLenum fail, GLenum zfail, GLenum zpass)`
- Activar:
  - `glEnable(GL_STENCIL_TEST)`

Stencil test

- Definir la función de test (cómo un fragmento pasa o no pasa el test)
- **void glStencilFunc(GLenum *func*, GLint *ref*, GLuint *mask*)**
  - Donde *func*: GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS
  - La operación se aplica del siguiente modo:

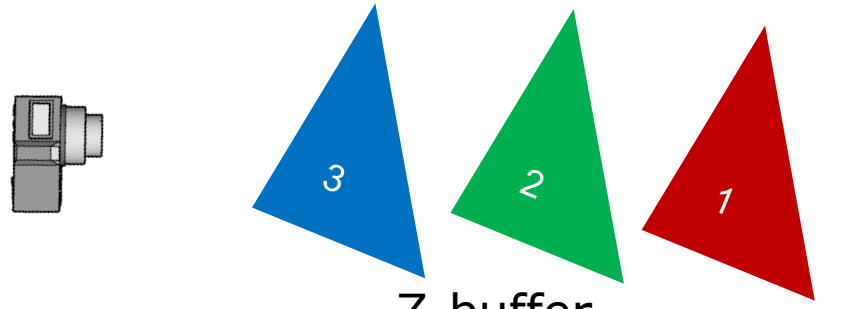
```
if (( ref & mask) <func> ( stencil & mask)) {
    Dibujar fragmento
    Actualizar stencil (Pasa/Falla Z)
} else {
    Descartar fragmento
    Actualizar stencil (Falla)
}
```

Stencil test

- Definir qué pasa con el valor del stencil buffer cuando el test pasa/falla
- **`void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass )`**
  - Donde:
    - *fail*: qué hacer cuando el stencil test falla, *zfail*: cuando stencil pasa pero depth no, *zpass*: cuando los dos pasan
  - cada argumento puede ser:
    - `GL_KEEP`: mantener el valor actual
    - `GL_ZERO`: poner el valor a cero
    - `GL_REPLACE`: reemplazarlo por el valor referencia
    - `GL_INCR`: incrementar hasta máximo
    - `GL_INCR_WRAP`: incremento circular
    - `GL_DECR`: decrementar hasta cero
    - `GL_DECR_WRAP`: decremento circular
    - `GL_INVERT`: invertir el valor bit a bit

Stencil test

```
                                        ref     mask
glStencilFunc(GL_GREATER, 2, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
                      fail      zfail      pass
```



Color

Z-buffer

1	1	1
1	1	1

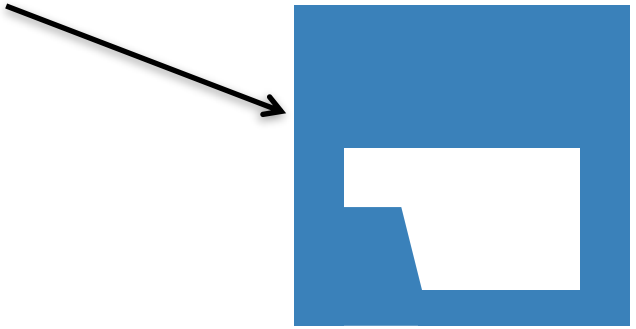
Stencil

00000000	00000000	00000000
00000000	00000000	00000000

Stencil test

- Cómo escribir en el stencil buffer
  - No se puede dibujar geometría al stencil buffer

```
Establecer el viewport y proyección para la plantilla()
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0x1, 0xFF);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
Dibujar la forma de la plantilla()
```



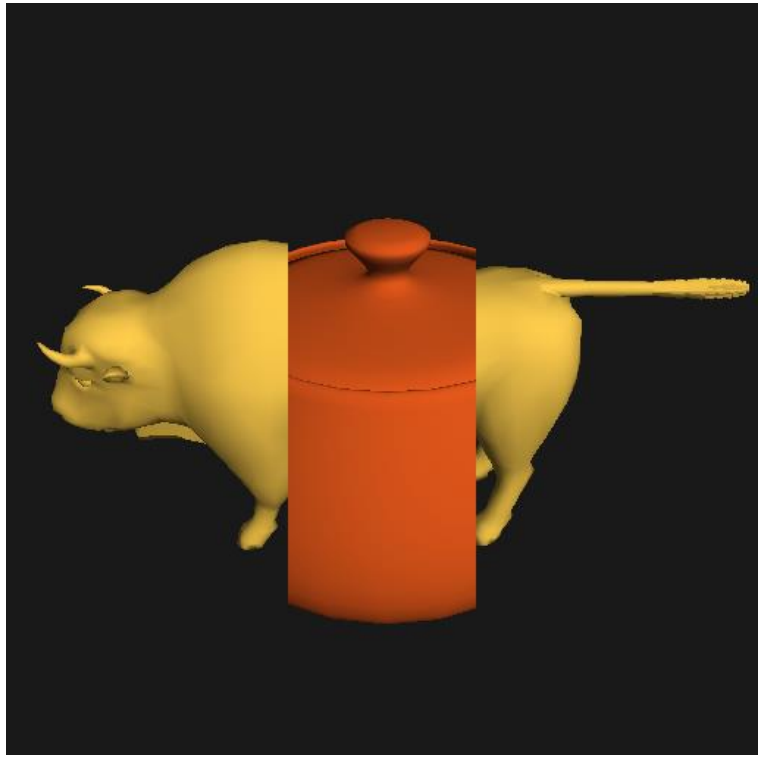
Stencil test

- Cómo usar el stencil buffer
  - Una vez que se tiene en el stencil buffer la máscara:

```
Establecer viewport y proyección para dibujar la escena()
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_EQUAL, 0x1, 0xFF);
/* Sólo se procesarán los píxeles cuyo valor de stencil sea 1 */
glStencilFunc(GL_NOTEQUAL, 0x1, 0xFF);
/* Sólo se procesarán los píxeles cuyo valor de stencil sea distinto de 1 */
```

  - Los valores del stencil no se modifican

Stencil test



Proyecto ej2-1

El *depth buffer*

- Se usa para implementar el algoritmo de visibilidad del Z-buffer
- Para cada pixel en pantalla, el depth buffer almacena la distancia normalizada (0..1) entre el plano de recorte frontal y la zona del polígono representada por el píxel
- Si el fragmento que se está procesando tiene una distancia menor a la que ya había, entonces se escribirán sus valores asociados a los buffers activos (por ejemplo, color y depth)

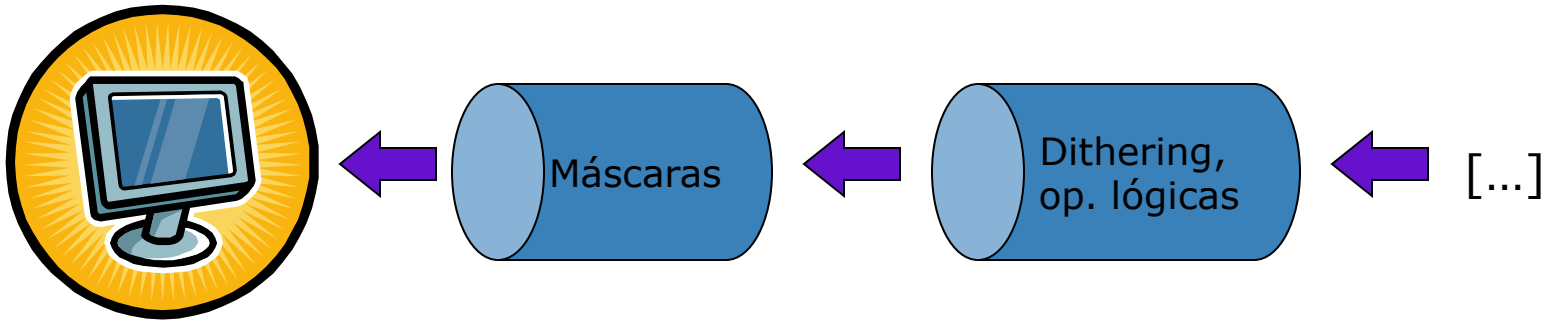
Depth test

- Reservar el buffer
  - **`myApp.initApp(argc, argv, PGUPV::DOUBLE_BUFFER | PGUPV::DEPTH_BUFFER | PGUPV::STENCIL_BUFFER);`**
- Activarlo
  - **`glEnable(GL_DEPTH_TEST)`**
- Se puede elegir la función de comparación:
  - **`void glDepthFunc( GLenum` *`func`* `);`**
  - donde *func*: `GL_NEVER`, `GL_ALWAYS`, `GL_LESS`, `GL_LEQUAL`, `GL_EQUAL`, `GL_GEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`

Escribiendo al buffer

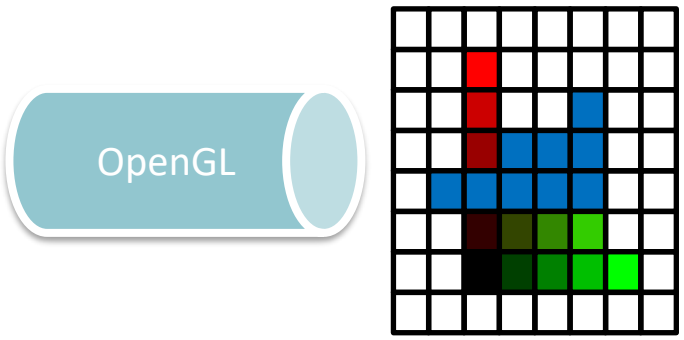
- Justo antes de escribir a cada buffer del framebuffer, OpenGL aplica unas máscaras:
  - **void glColorMask(GLboolean** *red*, **GLboolean** *green*, **GLboolean** *blue*, **GLboolean** *alpha* **);**
  - **void glDepthMask(GLboolean** *flag* **);**
  - **void glStencilMask(GLuint** *mask* **);** // máscara de bits
  - **void glStencilMaskSeparate(GLenum** face, **GLuint** *mask* **);**

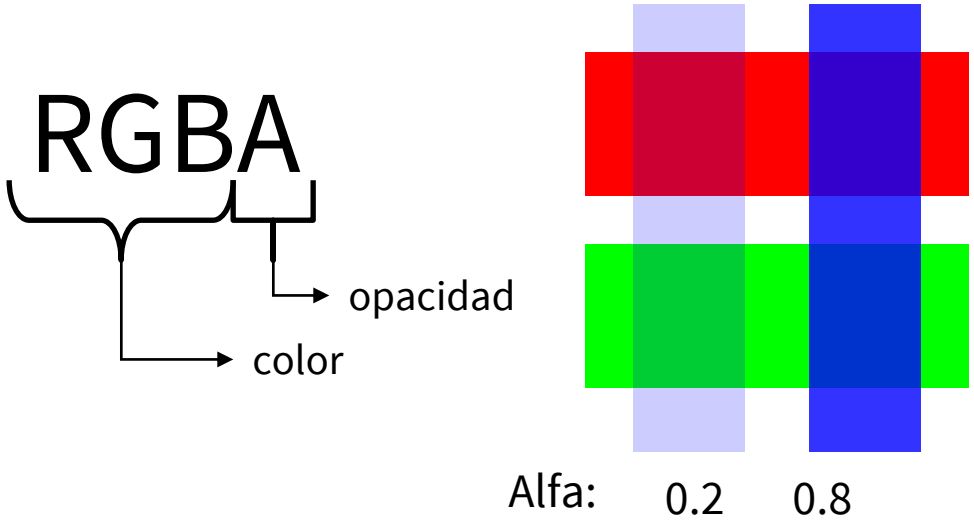
```
// face: GL_FRONT, GL_BACK, o GL_FRONT_AND_BACK
```



Blending

- Por defecto, cada polígono rasterizado por OpenGL sustituye el color que tenían los píxeles donde se dibuja
- Al activar el *blending,* en vez de sustituir el color existente, se mezclan
- Para la mezcla, normalmente se usa la componente alfa (RGB**A**) definida con cada color





Blending

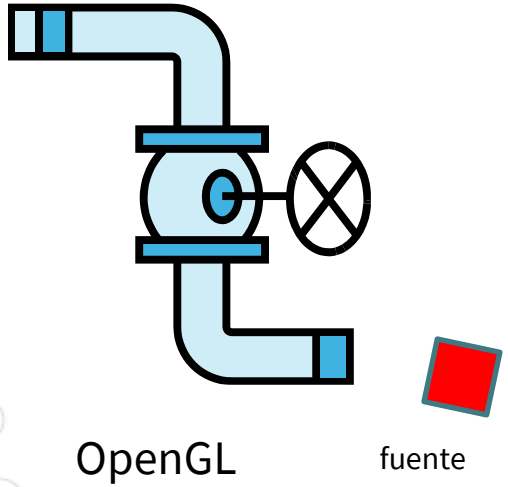
- El *blending* se puede usar para simular objetos translúcidos, reflejos, composición digital, chroma-key (blue-screen)…

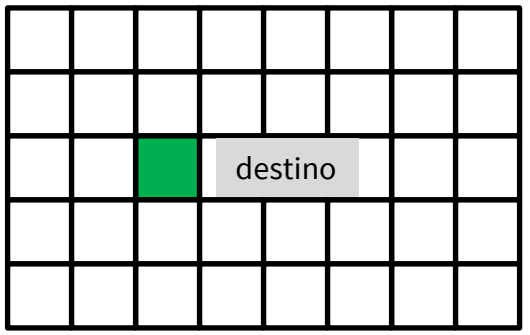
http://flickr.com/photos/ellie6

http://flickr.com/photos/82046831@N00/

Blending en OpenGL

- El programador puede seleccionar entre una serie de funciones predeterminadas para especificar cómo calcular el color final del pixel
- Conceptos:
  - Fragmento fuente: el fragmento que se acaba de generar (del polígono translúcido)
  - Píxel destino: píxel que ya se encuentra en el buffer de color





Blending en OpenGL

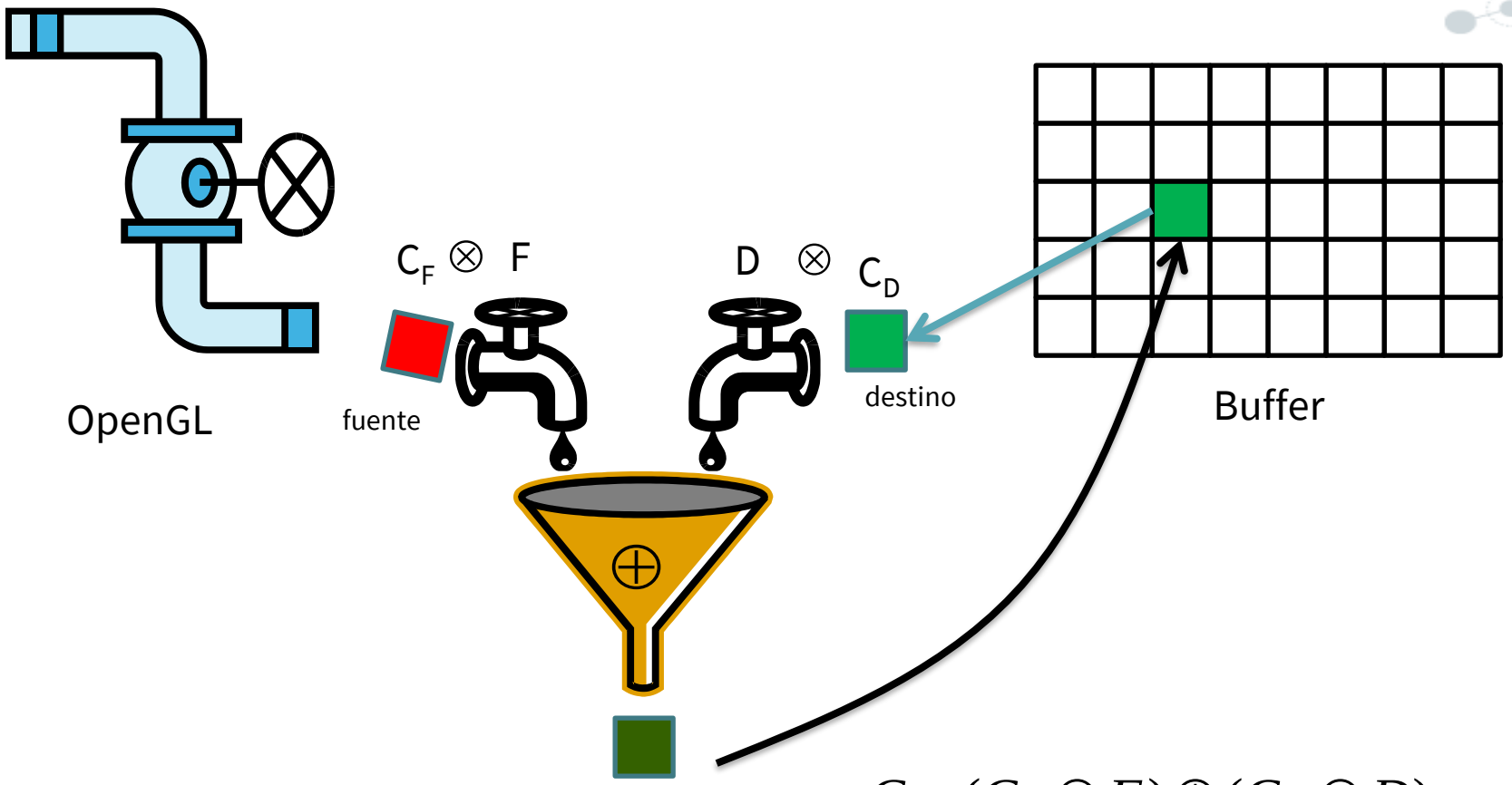
- Dos pasos:
  - Determinar los factores (pesos) de mezcla de la fuente ($F=\{F_R,F_G,F_B,F_A\}$) y el destino ($D=\{D_R,D_G,D_B,D_A\}$)
  - Combinar los colores fuente ($C_F=\{R_F,G_F,B_F,A_F\}$) y destino ($C_D=\{R_D,G_D,B_D,A_D\}$) y sus factores

$$C = (C_F \otimes F) \oplus (C_D \otimes D)$$

donde:

- $\otimes$ producto componente a componente
- $\oplus$ función seleccionable

Blending en OpenGL



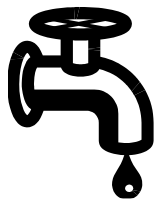
$$C = (C_F \otimes F) \oplus (C_D \otimes D)$$

Blending en OpenGL

Definiendo los factores de mezcla

- Para seleccionar los factores de mezcla:

F



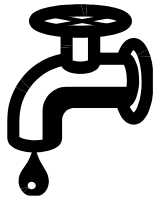
- **void glBlendFunc(GLenum** *sfactor*, **GLenum** *dfactor* **);**
- **void glBlendFuncSeparate(**

  GLenum *srcRGB*, **GLenum** *destRGB*,

  GLenum *srcAlpha*, **GLenum** *destAlpha* **);**

donde los parámetros indican cómo calcular cada factor, bien en global para RGBA, o bien por separado (RGB por un lado, y A por otro)

D



Blending en OpenGL

```
void glBlendColor(red, green, blue, alpha);
```

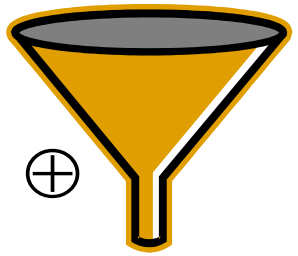
Constante	Factor RGB	Factor alfa
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(R_F, G_F, B_F)	A_F
GL_ONE_MINUS_SRC_COLOR	$(1-R_F, 1-G_F, 1-B_F)$	$1-A_F$
GL_DST_COLOR	(R_D, G_D, B_D)	A_D
GL_ONE_MINUS_DST_COLOR	$(1-R_D, 1-G_D, 1-B_D)$	$1-A_D$
GL_SRC_ALPHA	(A_F, A_F, A_F)	A_F
GL_ONE_MINUS_SRC_ALPHA	$(1-A_F, 1-A_F, 1-A_F)$	$1-A_F$
GL_DST_ALPHA	(A_D, A_D, A_D)	A_D
GL_ONE_MINUS_DST_ALPHA	$(1-A_D, 1-A_D, 1-A_D)$	$1-A_D$
GL_CONSTANT_COLOR	(R_C, G_C, B_C)	A_C
GL_ONE_MINUS_CONSTANT_COLOR	$(1-R_C, 1-G_C, 1-B_C)$	$1-A_C$
GL_CONSTANT_ALPHA	(A_C, A_C, A_C)	A_C
GL_ONE_MINUS_CONSTANT_ALPHA	$(1-A_C, 1-A_C, 1-A_C)$	$1-A_C$
GL_SRC_ALPHA_SATURATE	(f,f,f); $f=min(A_F, 1-A_D)$	1

También: GL_SRC1_COLOR, GL_ONE_MINUS_SRC1_COLOR, GL_SRC1_ALPHA, GL_ONE_MINUS_SRC1_ALPHA. El shader de fragmento genera dos colores (ver Sección 17.3.6.3 de la especificación de OpenGL 4.6).

Blending en OpenGL

Definiendo la función de mezcla

- Por defecto, para calcular el color del nuevo píxel, se suman los colores de origen y destino escalados mediante sus correspondientes factores:



$$C = (C_F \cdot F) + (C_D \cdot D) =$$
$$(R_F F_R + R_D D_R, G_F F_G + G_D D_G, B_F F_B + B_D D_B, A_F F_A + A_D D_A)$$

- Se puede usar otras funciones de mezcla:
  - **`void glBlendEquation(GLenum` *`mode`*`);`**
  - **`void glBlendEquationSeparate(GLenum` *`modeRGB`*`, GLenum` *`modeAlpha`* `);`**

    `mode: GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN ó GL_MAX`

Blending en OpenGL

Definiendo la función de mezcla

- Definición de la función de mezcla:

Modo	Operación
GL_FUNC_ADD	$C_F F + C_D D$
GL_FUNC_SUBTRACT	$C_F F - C_D D$
GL_FUNC_REVERSE_SUBTRACT	$C_D D - C_F F$
GL_MIN	$\min(C_F, C_D)$
GL_MAX	$\max(C_F, C_D)$

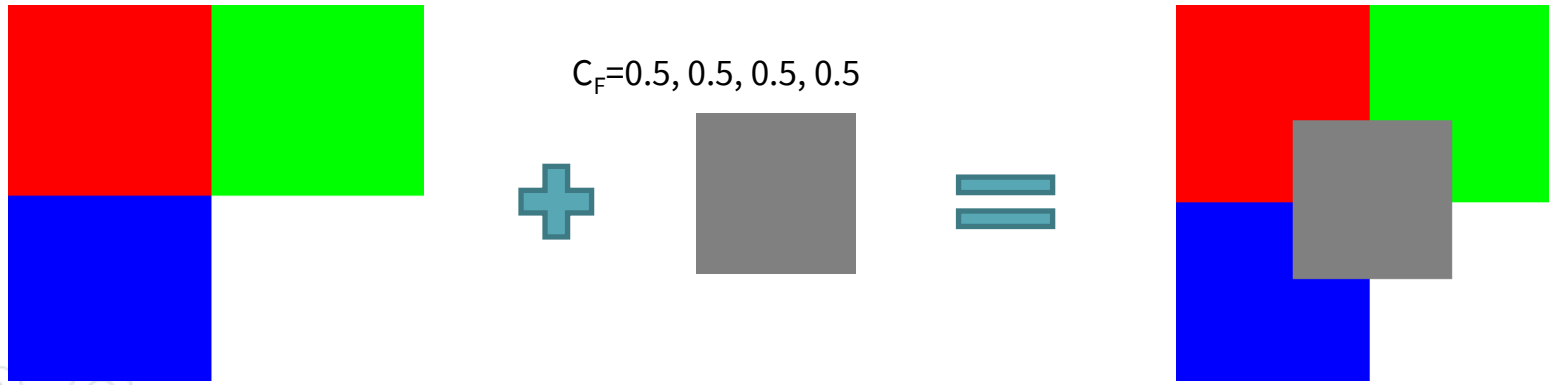
C_F: color fuente, C_D: color destino, F: factor de mezcla fuente, D: factor de mezcla destino

Blending en OpenGL

- Por defecto está desactivado:
  - **`glEnable(GL_BLEND)`**
  - ¡Cuidado!: normalmente no dibujarás todos los polígonos de la escena con *blending*
  - Para desactivar **`glDisable(GL_BLEND)`**

Blending en OpenGL

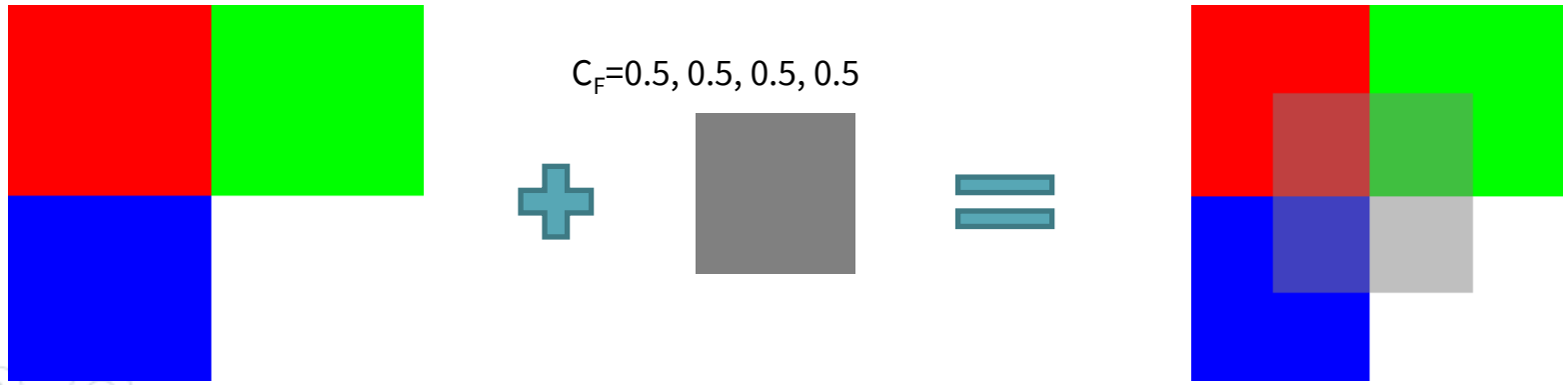
- Ejemplos de uso típicos
  - No hacer blending (reemplazar el color destino con el fuente):
    - F=GL_ONE, D=GL_ZERO, GL_FUNC_ADD
    - $C=C_F*1+C_D*0$



Proyecto ej2-2

Blending en OpenGL

- Ejemplos de uso típicos
  - Mezclar al 50% la fuente y el destino:
    - F=GL_SRC_ALPHA, D=GL_ONE_MINUS_SRC_ALPHA, GL_FUNC_ADD
    - Poner el alfa del polígono translúcido a 0.5
    - $C=C_F*0.5+C_D*(1-0.5)$
    - El porcentaje de la mezcla lo decide el alfa del polígono translúcido



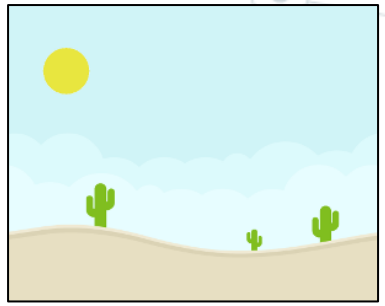
Blending en OpenGL

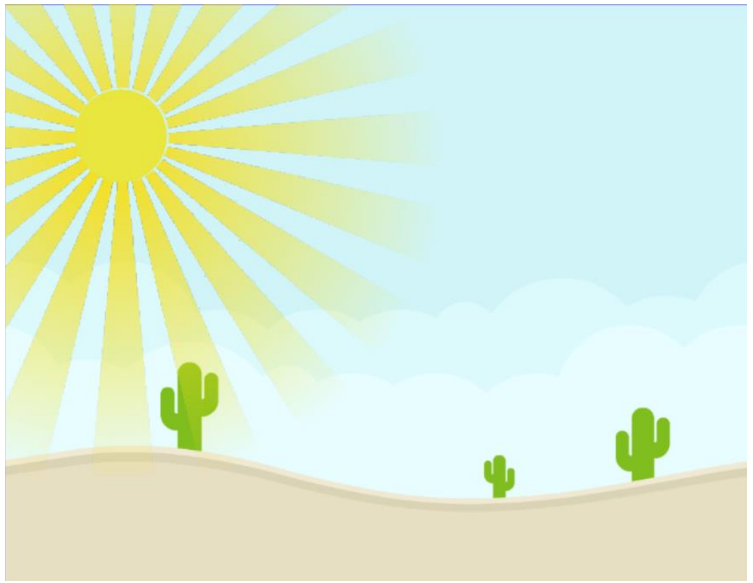
- Ejemplos de uso típicos
  - Mezclar al x% la fuente y el destino:
    - En vez de usar el alfa de los polígonos de entrada (sobre los que quizá no se tiene control), se puede utilizar un factor constante, p.e. `GL_CONSTANT_ALPHA`)
    - F=GL_CONSTANT_ALPHA, D=GL_ONE_MINUS_CONSTANT_ALPHA, GL_FUNC_ADD
    - Establecer el alfa constante con la función `glBlendColor`
    - $C=C_F*x+C_D*(1-x)$

Blending en OpenGL

- Ejemplos de uso típicos
  - Filtrar el color de una imagen dada según un porcentaje por canal:
    - F=GL_ZERO, D=GL_SRC_COLOR, GL_FUNC_ADD
    - Elegir el color del polígono fuente con los porcentajes deseados. P.e.: (0.2, 0.8, 0.0)
    - $C = C_F * 0 + C_D * (0.2, 0.7, 0.0)$
    - El resultado es el color de la imagen en el framebuffer, a la que se le ha eliminado el 80% del rojo, el 30% del verde y el 100% del azul

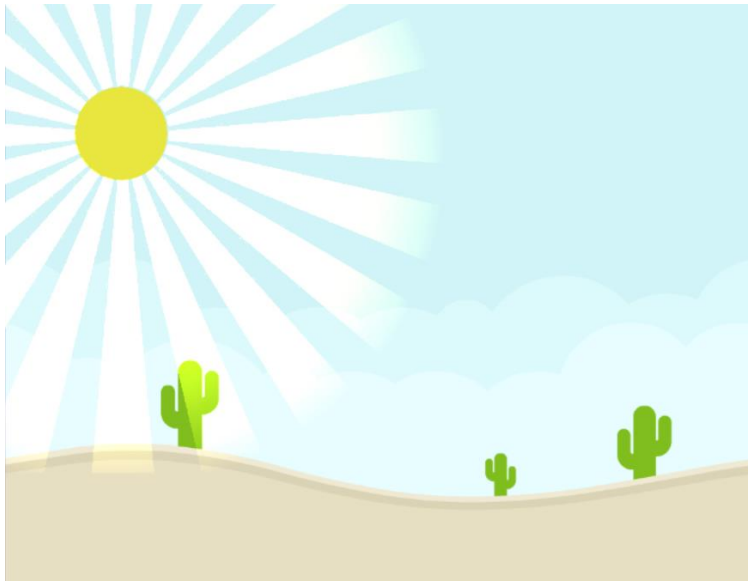
Blending en OpenGL





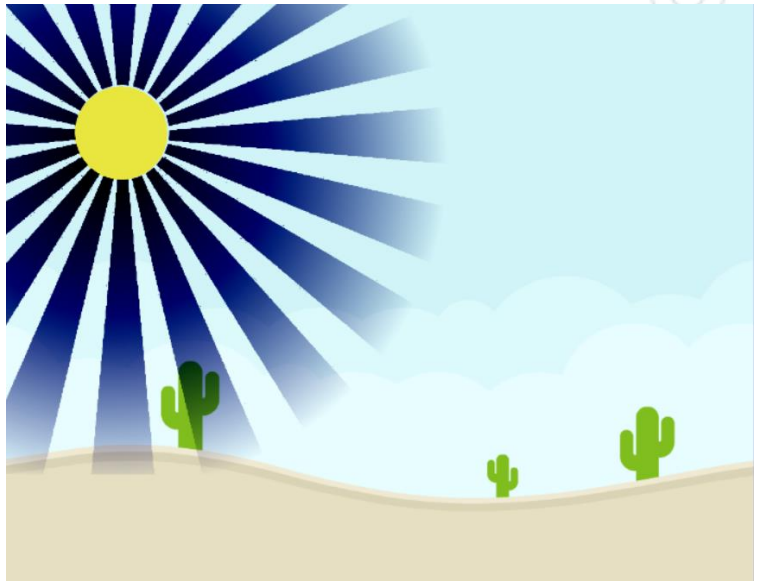
GL_SRC_ALPHA,
GL_ONE_MINUS_SCR_ALPHA

Blending aditivo



GL_SRC_ALPHA, GL_ONE

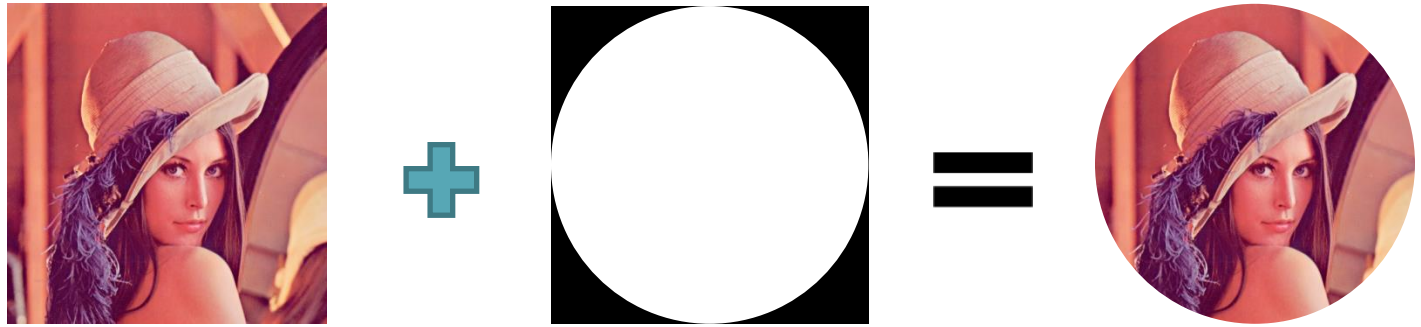
Blending substractivo



GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA,
GL_FUNC_REVERSE_SUBTRACT

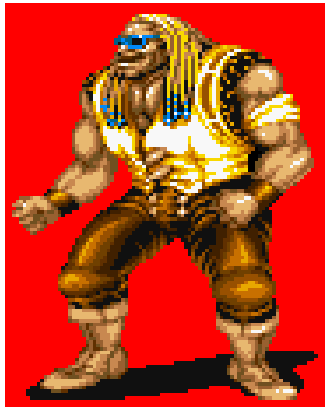
Blending en OpenGL

- Otros ejemplos
  - Se usa en texturas para mostrar imágenes no rectangulares. Cada píxel define su propio alfa. Aquellos píxeles con alfa=0 no se dibujan (*alpha testing*).
  - También se usa para técnicas de antialiasing.



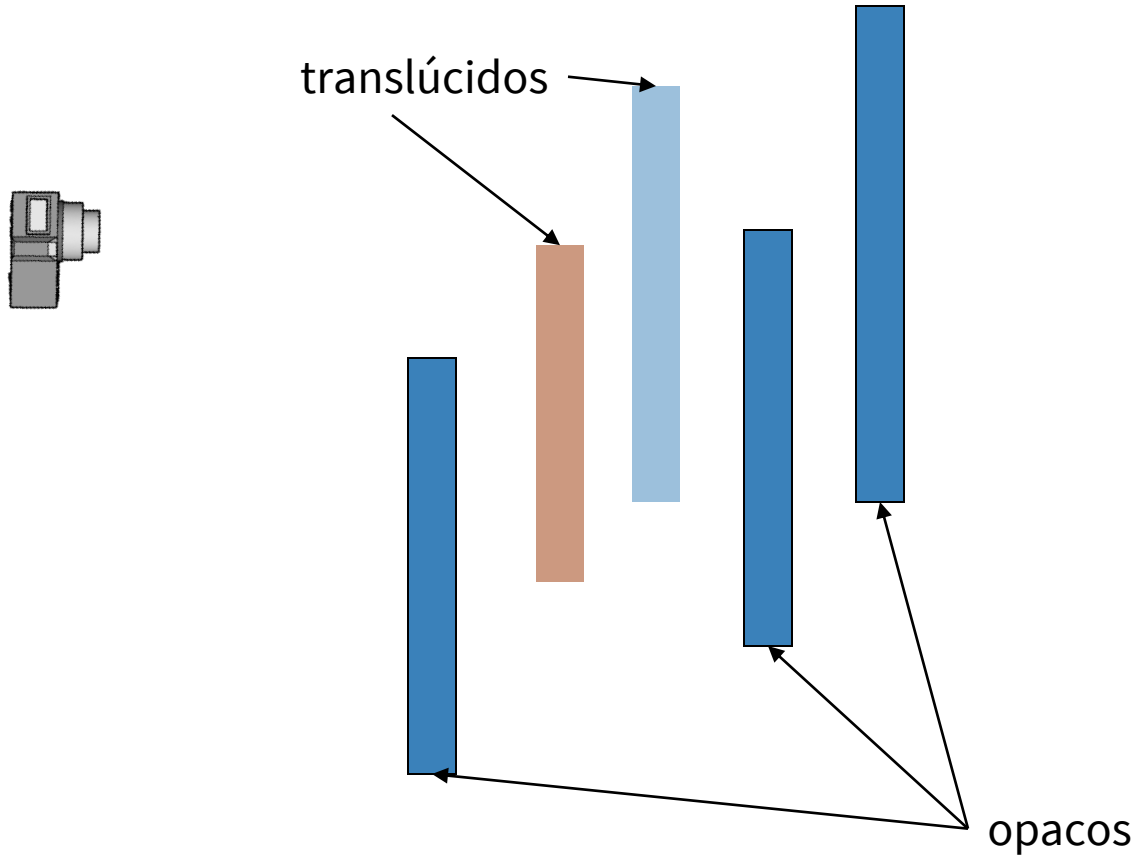
Blending en OpenGL

- Juegos 2D basados en sprites:



Blending en OpenGL

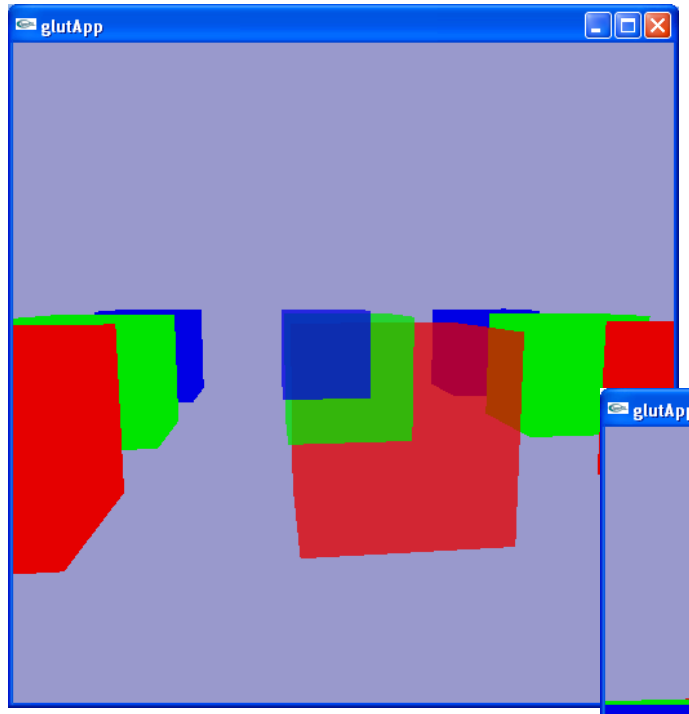
- El orden de dibujo es importante.



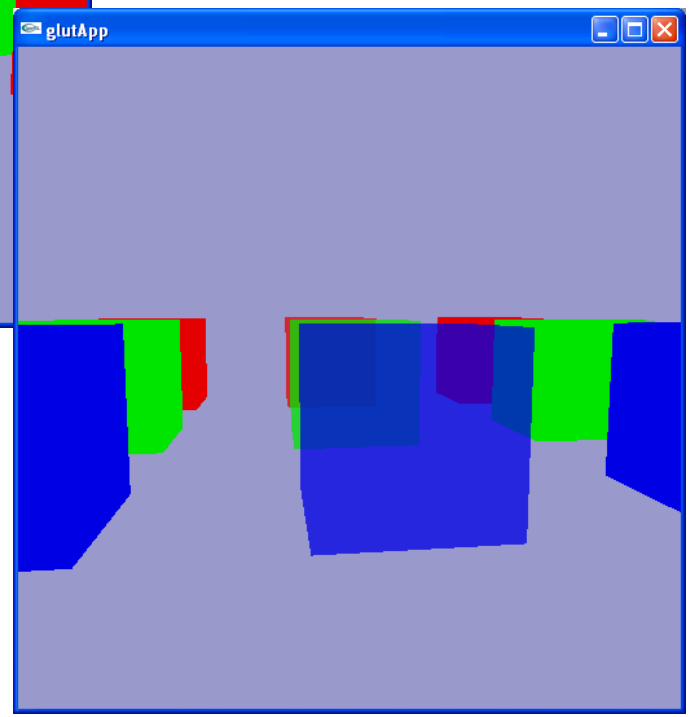
Blending en OpenGL

- ¡Cuidado!
  - El orden de los *factores* sí altera el producto.
  - Para dibujar una escena con varias superficies translúcidas:
    - Dibujar todos los objetos opacos
    - Desactivar la escritura al Z-buffer
      - `glDepthMask(GL_FALSE);`
    - Dibujar los objetos translúcidos
    - Activar la escritura al Z-buffer
      - `glDepthMask(GL_TRUE);`
  - El mejor resultado se consigue cuando se dibujan los objetos translúcidos de *atrás hacia adelante*

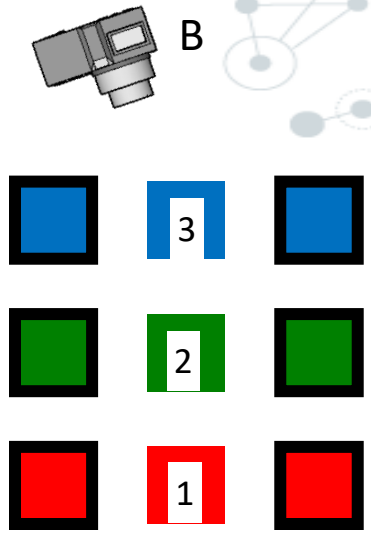
Blending en OpenGL



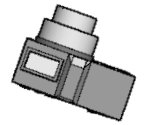
A



B



B

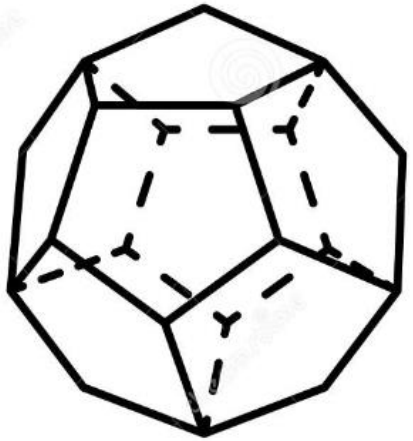


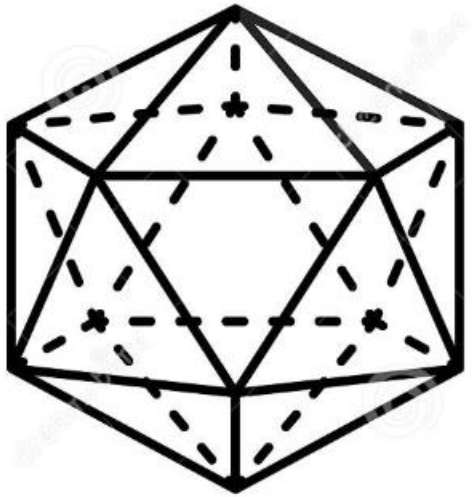
A

Proyecto ej2-3

Back face culling

- Si todos los objetos de la escena son opacos y cerrados
  - Ninguna cara trasera es visible
  - Podríamos evitar dibujarlas
- Aparte del posible aumento de velocidad, se usa a veces para procesar de distinta forma los polígonos dependiendo de su orientación a la cámara





Back face culling

- Funciones asociadas al *culling*:
  - Definir la ordenación de los vértices de la cara frontal en pantalla

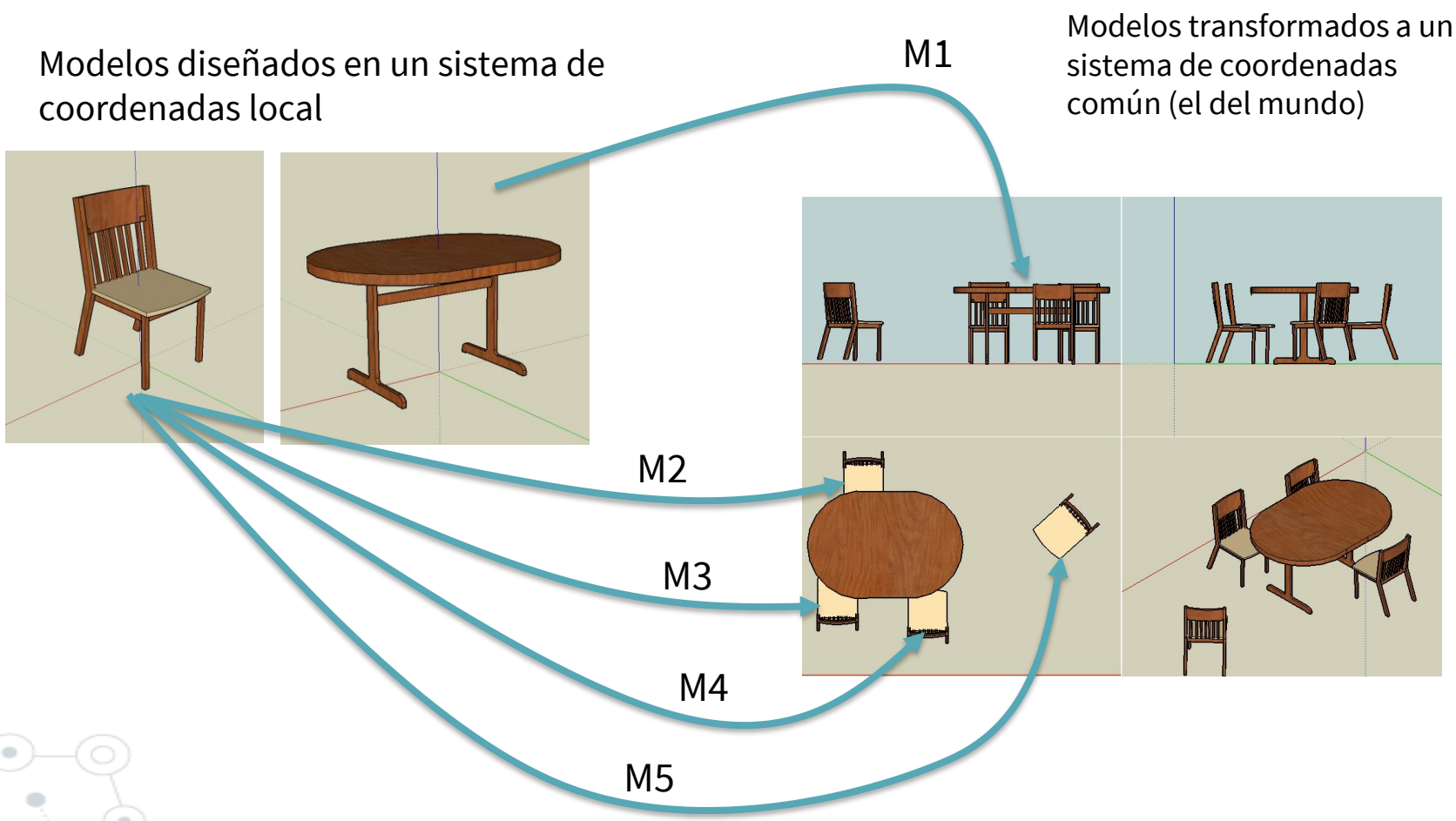
    `glFrontFace(GL_CW | GL_CCW)`
  - Qué cara descartar

    `glCullFace(GL_FRONT | GL_BACK | GL_FRONT_AND_BACK)`
  - Activar el culling

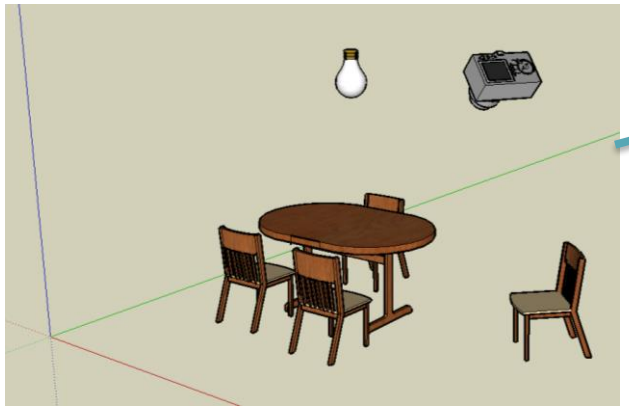
    `glEnable(GL_CULL_FACE)`
  - Desactivar el culling

    `glDisable(GL_CULL_FACE)`

La tubería de *rendering* de vértices



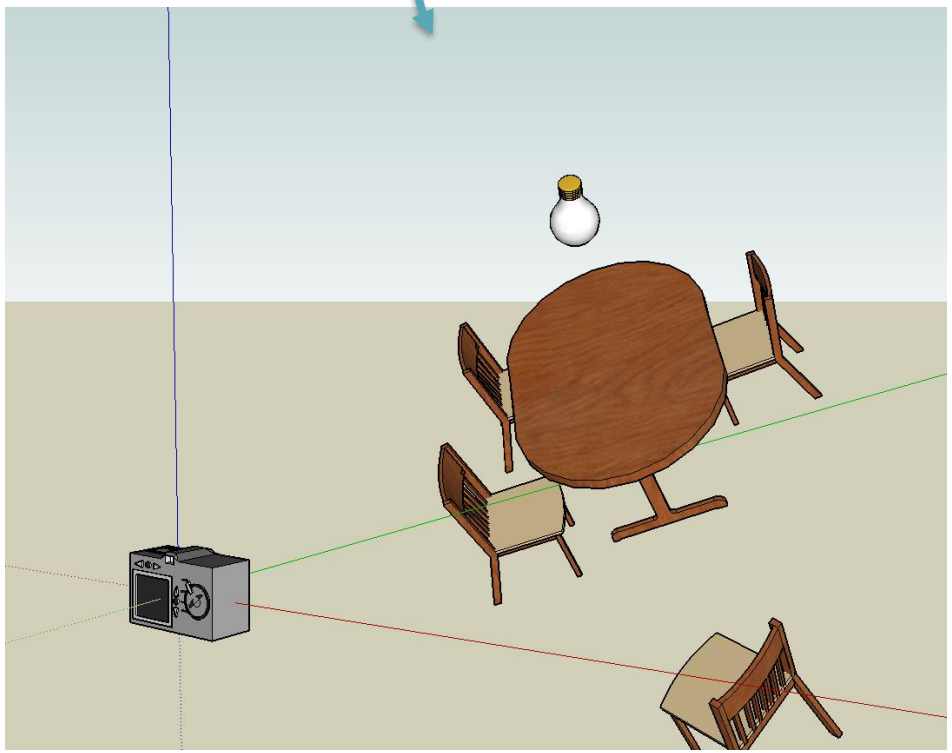
La tubería de *rendering* de vértices



La posición de la cámara también se define en el sistema de coordenadas del mundo

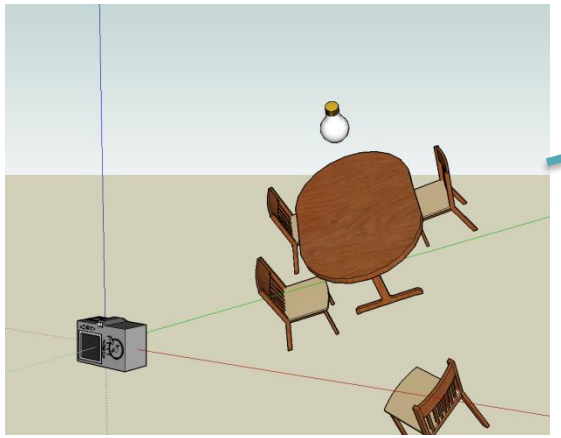
V

Todo está definido en el sistema de coordenadas de la cámara (*eye space*)



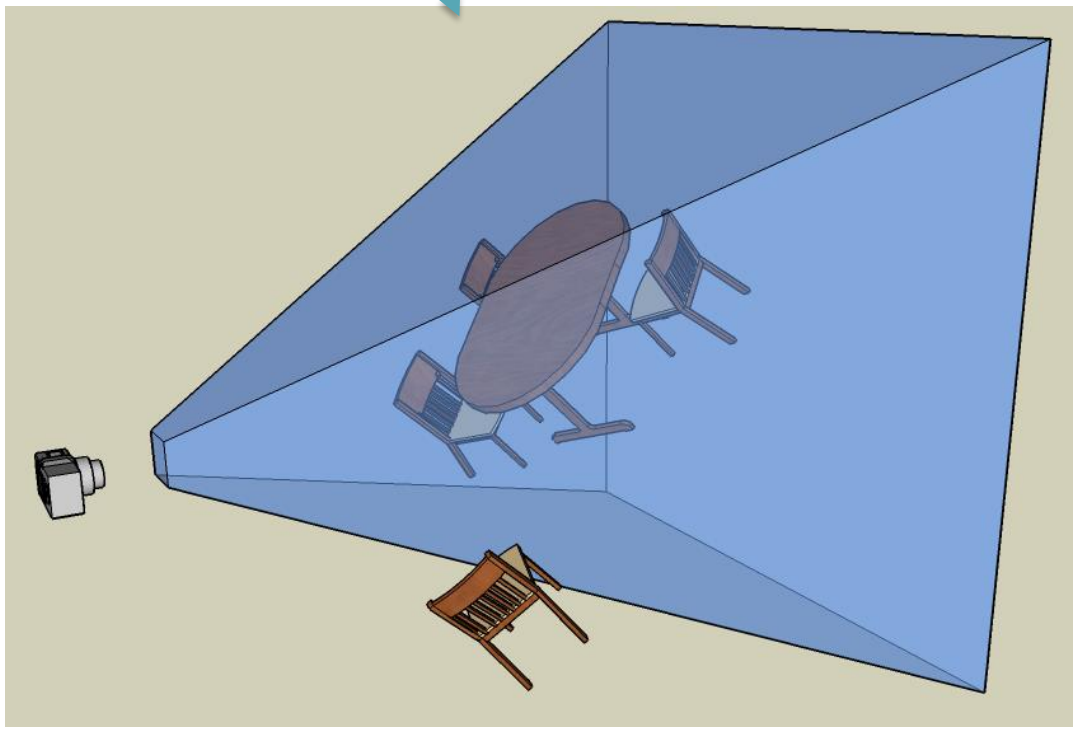
Para ahorrar transformaciones, las luces se definen normalmente en el sistema de coordenadas de la cámara

La tubería de *rendering* de vértices



P

La matriz *projection* define el volumen de la vista

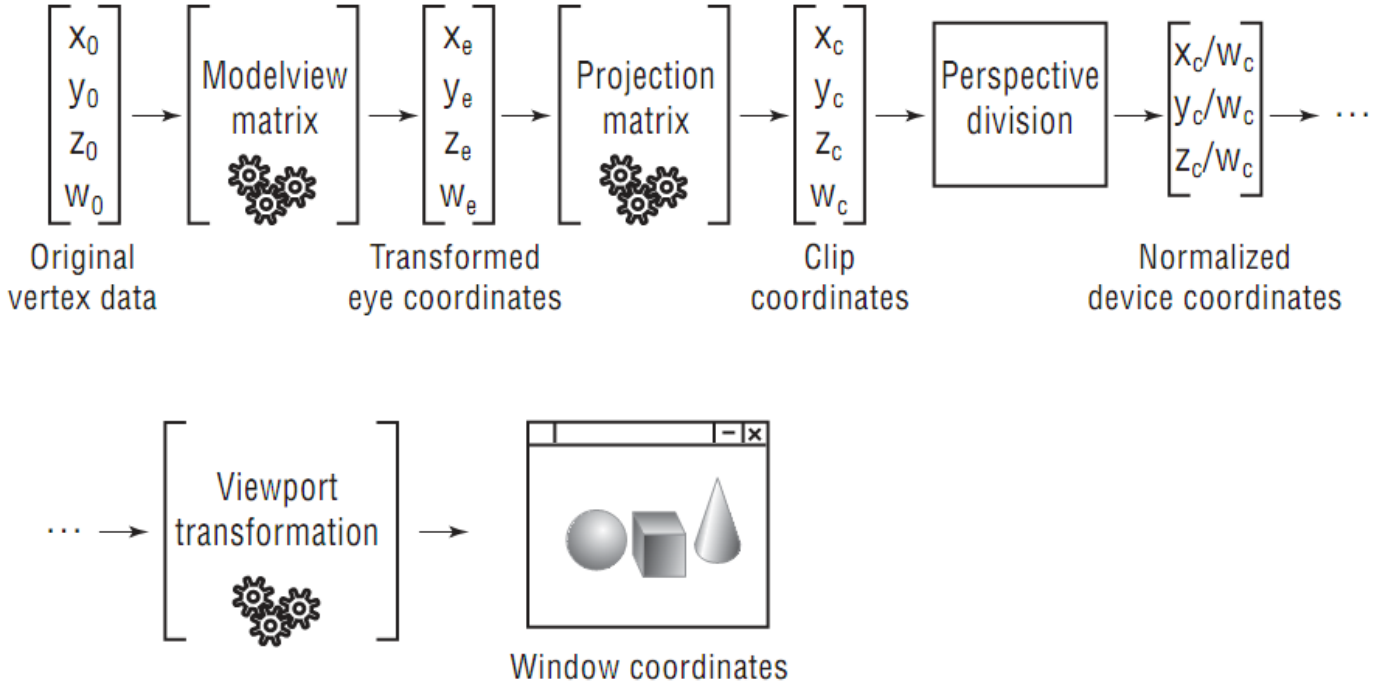


Dos tipos de cámaras en OpenGL:

- perspectivas
- paralelas

La tubería de *rendering* de vértices

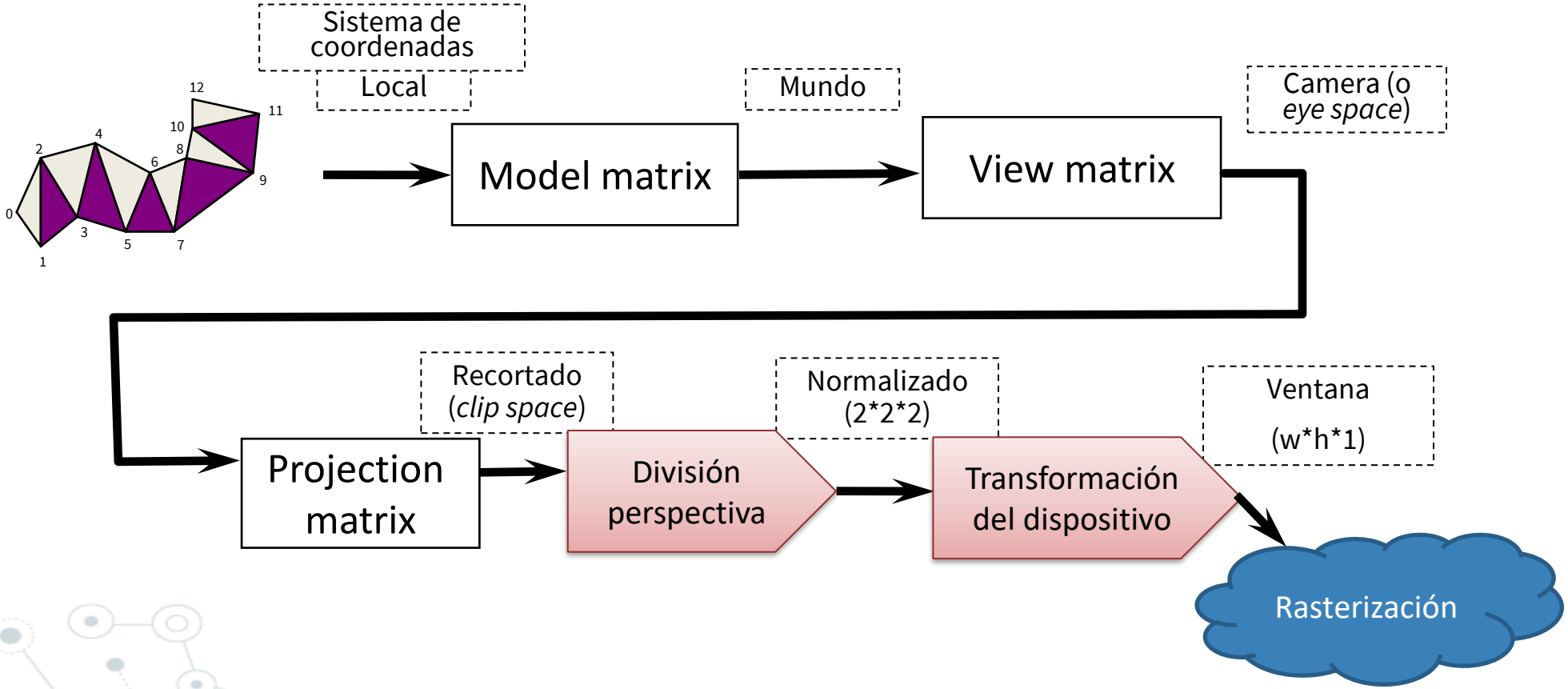
- La tubería tradicional de OpenGL



Superbiblia 5ªed, pp. 159

La tubería de *rendering* de vértices

- La tubería que usaremos nosotros (reimplementada en CPU)



Re-implementación del procesado de vértices en PGUPV

- Se han definido varias clases para facilitar la definición y aplicación de las matrices
  - **`MatrixStack`**: pila de matrices, con las operaciones de manipulación de matrices
  - **`GLMatrices`**: conjunto de pilas de matrices que (re)implementa la tubería de OpenGL extendida

Re-implementación del procesado de vértices en PGUPV

```
namespace PGUPV {
  class MatrixStack {
  public:
    MatrixStack();
    void loadIdentity();
    void pushMatrix();
    void popMatrix();
    void translate(float x, float y, float z);
    void translate(const glm::vec3 &t);
    void rotate(float radians, float axis_x, float axis_y, float axis_z);
    void rotate(float radians, const glm::vec3 &axis);
    void scale(float sx, float sy, float sz);
    void scale(const glm::vec3 &s);
    void multMatrix(const glm::mat4 &m);
    void setMatrix(const glm::mat4 &m);
    const glm::mat4 &getMatrix() const;
    void reset();
  private:
    std::vector<glm::mat4> matrix;
  };
};
```

```
PGUPV::MatrixStack m;

cout << "Matriz identidad: " << endl;
m.loadIdentity();
cout << m.getMatrix() << endl;

cout << "Matriz que traslada 10 unidades en +X" << endl;
m.translate(10.0f, 0.0f, 0.0f);
cout << m.getMatrix() << endl;

glm::vec4 p = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);
cout << "p= " << p << endl;

cout << "M*p = " << m.getMatrix() * p << endl;

cout << "pushMatrix()" << endl;
m.pushMatrix();
m.rotate(glm::radians(90.0f), 0.0f, 1.0f, 0.0f);

cout << "Ahora M = T(10, 0, 0)*R(PI/2, 0, 1, 0)" << endl;
cout << "M=" << endl << m.getMatrix() << endl;

p=glm::vec4(1.0f, 0.0f, 0.0f, 1.0f);
cout << "p= " << p << endl;

cout << "M*p = " << m.getMatrix() * p << endl;

cout << "popMatrix()" << endl;
m.popMatrix();
cout << "M=" << endl << m.getMatrix() << endl;
```

```
Matriz identidad:
1.0, 0.0, 0.0, 0.0
0.0, 1.0, 0.0, 0.0
0.0, 0.0, 1.0, 0.0
0.0, 0.0, 0.0, 1.0

Matriz que traslada 10 unidades en +X:
1.0, 0.0, 0.0, 10.0
0.0, 1.0, 0.0, 0.0
0.0, 0.0, 1.0, 0.0
0.0, 0.0, 0.0, 1.0

p= (1.0, 1.0, 1.0, 1.0)
M*p = (11.0, 1.0, 1.0, 1.0)
pushMatrix()
Ahora M = T(10, 0, 0)*R(PI/2, 0, 1, 0)
M=
-0.0, 0.0, 1.0, 10.0
0.0, 1.0, 0.0, 0.0
-1.0, 0.0, -0.0, 0.0
0.0, 0.0, 0.0, 1.0

p= (1.0, 0.0, 0.0, 1.0)
M*p = (10.0, 0.0, -1.0, 1.0)
popMatrix()
M=
1.0, 0.0, 0.0, 10.0
0.0, 1.0, 0.0, 0.0
0.0, 0.0, 1.0, 0.0
0.0, 0.0, 0.0, 1.0
```

Re-implementación del procesado de vértices en PGUPV

```
class GLMatrices : public UniformBufferObject {
public:
  // ¡Cuidado! Las matrices ModelView y Normal se actualizan automáticamente
  // a partir de las demás: NO modificarlas
  enum Matrix {MODEL_MATRIX, VIEW_MATRIX, PROJ_MATRIX, MODELVIEW_MATRIX, NORMAL_MATRIX};
[…]
  void loadIdentity(Matrix mat);
  void pushMatrix(Matrix mat);
  void popMatrix(Matrix mat);
  void translate(Matrix mat, float x, float y, float z);
  void translate(Matrix mat, const glm::vec3 &t);
  void rotate(Matrix mat, float radians, float axis_x, float axis_y, float axis_z);
  void rotate(Matrix mat, float radians, const glm::vec3 &axis);
  void scale(Matrix mat, float sx, float sy, float sz);
  void scale(Matrix mat, const glm::vec3 &s);
  void multMatrix(Matrix mat, const glm::mat4 &m);
  void setMatrix(Matrix mat, const glm::mat4 &m);
  const glm::mat4 &getMatrix(Matrix mat) const;
  const glm::mat3 getNormalMatrix() const;
  […]
};
```

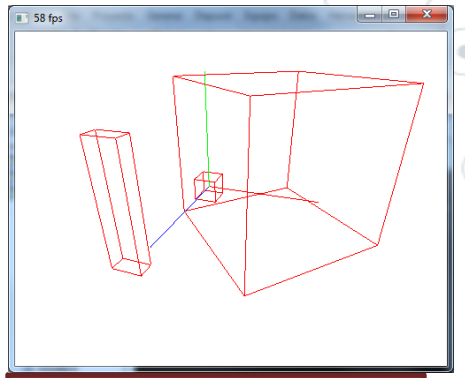
Re-implementación del procesado de vértices en PGUPV

```
void MyRender::render() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  mats.setMatrix(GLMatrices::VIEW_MATRIX, getCamera().getViewMatrix());
  shader.use();
  axes.render();

  mats.pushMatrix(GLMatrices::MODEL_MATRIX);
  mats.scale(GLMatrices::MODEL_MATRIX, 0.2f, 0.2f, 0.2f);
  box.render();
  mats.popMatrix(GLMatrices::MODEL_MATRIX);

  mats.pushMatrix(GLMatrices::MODEL_MATRIX);
  mats.translate(GLMatrices::MODEL_MATRIX, 0.7f, 0.5f, 0.5f);
  mats.rotate(GLMatrices::MODEL_MATRIX, glm::radians(45.f), 0.0f, 1.0f, 0.0f);
  box.render();
  mats.popMatrix(GLMatrices::MODEL_MATRIX);

  mats.pushMatrix(GLMatrices::MODEL_MATRIX);
  mats.translate(GLMatrices::MODEL_MATRIX, 0.0f, 0.4f, 1.2f);
  mats.scale(GLMatrices::MODEL_MATRIX, 0.2f, 0.8f, 0.1f);
  box.render();
  mats.popMatrix(GLMatrices::MODEL_MATRIX);
}
```



Proyecto ej2-5

Biblioteca OpenGL Mathematics (GLM)

- http://glm.g-truc.net
- Es una biblioteca *open source* que ofrece una implementación en la CPU de las funciones matemáticas disponibles en GLSL
- Es una biblioteca con sólo cabeceras
- También ofrece funciones para transformación de matrices, cuaterniones, números aleatorios, etc.
- Documentación en **`pract-pg\librerias\glm\doc`**

Biblioteca OpenGL Mathematics (GLM)

- Includes:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

- Namespace `glm`:
  - Opciones de uso:

```
#include <glm/glm.hpp>

glm::vec4 v;
[…]
v=glm::abs(v);
```

```
#include <glm/glm.hpp>
using glm::vec4;
using glm::abs;

vec4 v;
[…]
v = abs(v);
```

```
#include <glm/glm.hpp>
using namespace glm;

vec4 v;
[…]
v = abs(v);
```

Biblioteca OpenGL Mathematics (GLM)

- Tipos definidos:
  - Coinciden con los de GLSL (ver tarjeta de referencia, “Transparent Types”)
  - **`glm::vec2, vec3, vec4, mat2, mat3, mat4, mat<C>x<F>, ivec2…, uvec2…, bvec2…, dvec2…`**

Biblioteca OpenGL Mathematics (GLM)

- Funciones:
  - Básicas
    - `glm::radians`, `length`, `distance`, `cross`, `dot`, y en general, las funciones predefinidas de GLSL (ver la tarjeta de referencia)

Biblioteca OpenGL Mathematics (GLM)

- Definición del volumen de la vista
  - `mat4 perspective(float fovy, float aspect, float zNear, float zFar)`
    - Sustituye a `gluPerspective`
  - `mat4 ortho(float left, float right, float bottom, float top, float zNear, float zFar)`
    - Sustituye a `glOrtho`
  - `mat4 ortho(left, right, bottom, top)`
    - Sustituye a `gluOrtho2D`
  - `mat4 frustum(left, right, bottom, top, nearVal, farVal)`
    - Sustituye a `glFrustum`

Biblioteca OpenGL Mathematics (GLM)

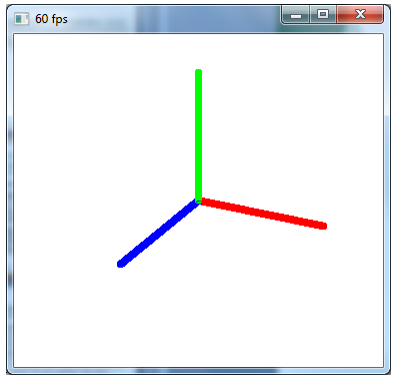
- Definición de la posición y orientación de la cámara
  - **`mat4 lookAt(vec3 eye, vec3 center, vec3 up)`**
    - Sustituye a **`gluLookAt`**

Biblioteca OpenGL Mathematics (GLM)

- Ejemplos:

```
void MyRender::reshape(uint w, uint h) {
  glViewport(0, 0, w, h);
  if (h == 0) h = 1;
  float ar = (float)w/h;
  mats.setMatrix(GLMatrices::PROJ_MATRIX,
    glm::perspective(glm::radians(60.0f), ar, .1f, 10.0f));
  mats.loadIdentity(GLMatrices::MODEL_MATRIX);
}
void MyRender::render() {
  glClear(GL_COLOR_BUFFER_BIT);
  mats.setMatrix(GLMatrices::VIEW_MATRIX,
     glm::lookAt(vec3(1, 1, 2), vec3(0, 0, 0), vec3(0, 1, 0)));
  shader.use();
  axes.render();
}
```

Proyecto ej1-3



Biblioteca OpenGL Mathematics (GLM)

- Transformaciones
  - `mat4 translate(mat4 const &m, vec3 v)`
  - `mat4 rotate(mat4 const &m, float angle, vec3 v)`
  - `mat4 scale(mat4 const &m, vec3 v)`
- Ejemplo:

```
glm::mat4 m;
m = translate(glm::mat4(1.0f), glm::vec3(10.f, 0.f, 0.f));
m = rotate(m, glm::pi()/4.0f, glm::vec3(0.0f, 1.0f, 0.0f));
m = scale(m, glm::vec3(1.0f, 2.0f, 1.0f));
glm::vec4 p, q;
q = m*p;
```

Biblioteca OpenGL Mathematics (GLM)

- Transformación de un vértice:

```
glm::mat4 m;
glm::vec4 v;
m = translate(glm::mat4(1.0f), glm::vec3(10.f, 0.f, 0.f));
v = m * v;
```

- Transformación de normales:
  - No se puede aplicar directamente la matriz *modelview*, sino la transpuesta de su inversa:

```
glm::vec3 n;
glm::mat3 nm = glm::transpose(glm::inverse(mat3(mvm)));
n = nm * n;
```