Introducción a OpenGL

Informática Gráfica I

Material de: **Ana Gil Luezas**
Adaptado por: **Elena Gómez** y **Rubén Rubio**
{mariaelena.gomez,rubenrub}@ucm.es

UNIVERSIDAD COMPLUTENSE MADRID

Contenido

¿Qué es OpenGL?

- **OpenGL** (*Open Graphics Library*) es una API (*Application Programming Interface*) portable que permite la comunicación entre el programador de aplicaciones y el hardware gráfico de la máquina o GPU (*Graphics Processing Unit*).
- Es una especificación gestionada actualmente por Khronos Group e implementada por los fabricantes de **GPUs** (que forman parte de ese consorcio).

¿Cómo funciona OpenGL?

OpenGL es una **máquina de estados**:

- Colección de variables de estado a las que se cambia su valor y se renderiza sobre el estado actual.
- El estado se conoce como **OpenGL context**.

¡Cuidado!

No es lo mismo dibujar un triángulo y activar una textura, que activar una textura y dibujar un triángulo.

¿Cómo funciona OpenGL?

La manera (sencilla) de dibujar en OpenGL:

1. Activar todas las opciones que van a ser persistentes a la escena (ponemos la cámara, activamos la iluminación global,...).
2. Activar las opciones que establecen el estado de un objeto específico (su posición en el espacio, su textura,...).
3. Dibujar el objeto.
4. Desactivar las opciones propias de ese objeto (volver a la posición original, desactivar su textura).
5. Volver al punto 2.

OpenGL SDK

Limitaciones:

- No existen comandos de alto nivel para cargar imágenes o describir escenas 3D.
- Tampoco dispone de comandos para gestionar ventanas ni para interactuar con el usuario.

OpenGL SDK

Bibliotecas:

- **GLFW** (*Graphics Library Framework*): biblioteca ligera y portable para la gestión de ventanas compatible con OpenGL, OpenGL ES y Vulkan.
- **GLM** (*OpenGL Mathematics*): biblioteca para las operaciones matemáticas especializada para la programación gráfica.
- **GLEW** (*OpenGL Extension Wrangler*): biblioteca para detectar y cargar extensiones de OpenGL.
- Otras utilidades: `stb_image`, **GL Image**, **GL Load**, ...

Sintaxis de los comandos

- Todos los comandos OpenGL comienzan con `gl`, y cada una de las palabras que componen el comando comienzan por letra mayúscula (*CamelCase*).

```
glClearColor(....)
glEnable(...)
```

- Las constantes se escriben en mayúsculas, y comienzan por `GL`. Cada una de las palabras que componen el identificador está separada de la anterior por _ (SNAKE_CASE).

```
GL_DEPTH_TEST
GL_COLOR_BUFFER_BIT
```

Sintaxis de los comandos

- Existen comandos en OpenGL que admiten distinto número y tipos de argumentos.
- Estos comandos terminan con el sufijo que indica el tipo de los mismos.

```
glTexParameteri(...)        // 1 int
glColor3d(GLdouble red,...) // 3 double
glColor4fv(GLfloat *)       // 4 float*
```

👉 **4fv**: indica que el parámetro es un puntero a un array de 4 *floats*.

```
glUniformMatrix4fv(..., const GLfloat* m)
glUniformMatrix3fv(..., const GLfloat* m)
```

👉 **Matrix4fv**: indica que los parámetros son punteros a un array de 4×4 *floats*.

Tipos básicos de OpenGL

- OpenGL trabaja internamente con tipos básicos específicos que son compatibles con los de C/C++, además de `GLboolean` (`GL_TRUE` / `GL_FALSE`).

Sufijo	Tipo OpenGL
b	`GLbyte` (entero de 8 bits)
ub	`GLubyte` (entero sin signo de 8 bits)
s	`GLshort` (entero con signo de 16 bits)
us	`GLushort` (entero sin signo de 16 bits)
i	`GLint` (entero de 32 bits)
ui	`GLuint`, `GLsizei`, `GLenum` (entero sin signo de 32 bits)
f	`GLfloat`, `GLclampf` (punto flotante de 32 bits)
d	`GLdouble`, `Glclampd` (punto flotante de 64 bits)

Tipos de datos de GLM

- **GLM** ofrece tipos, clases y funciones compatibles con OpenGL, GLSL (*OpenGL Shading Language*) y C++.
- Define el espacio de nombres **glm** y tipos para vectores y matrices:

```
glm::vec2, glm::vec3, glm::vec4
glm::dvec2, glm::ivec3, glm::uvec4, glm::bvec3
glm::mat4, glm::dmat4, glm::mat3, glm::dmat3
```

- Operaciones: *, +, ...

Tipos de datos de GLM

- Para las coordenadas de los vértices de las primitivas gráficas, se usan vectores de tipo `glm::vec3` (componentes: `v.x`, `v.y`, `v.z`).
- Para las componentes de los colores RGBA, se usan vectores de tipo `glm::vec4` (componentes `c.r`, `c.g`, `c.b`, `c[0]`, `c[1]`, `c[2]`).
- Para las matrices, se usan `glm::mat4` m, donde `m[i]` es la columna i-ésima de tipo (`vec4`).

Ventana

- El color de fondo de la **ventana** en la que se dibuja se puede modificar utilizando el comando:

```
glClearColor(GLfloat r,
  GLfloat g,
  GLfloat b,
  GLfloat alpha)
```

Los valores de los argumentos están en $[0, 1]$.

Ejemplo

Para poner el color de fondo negro:

```
glClearColor(0.0,0.0,0.0,0.0); //valores por defecto
```

Frame buffer

- Función `display()` de la ventana con doble buffer: **Front** y **Back**:

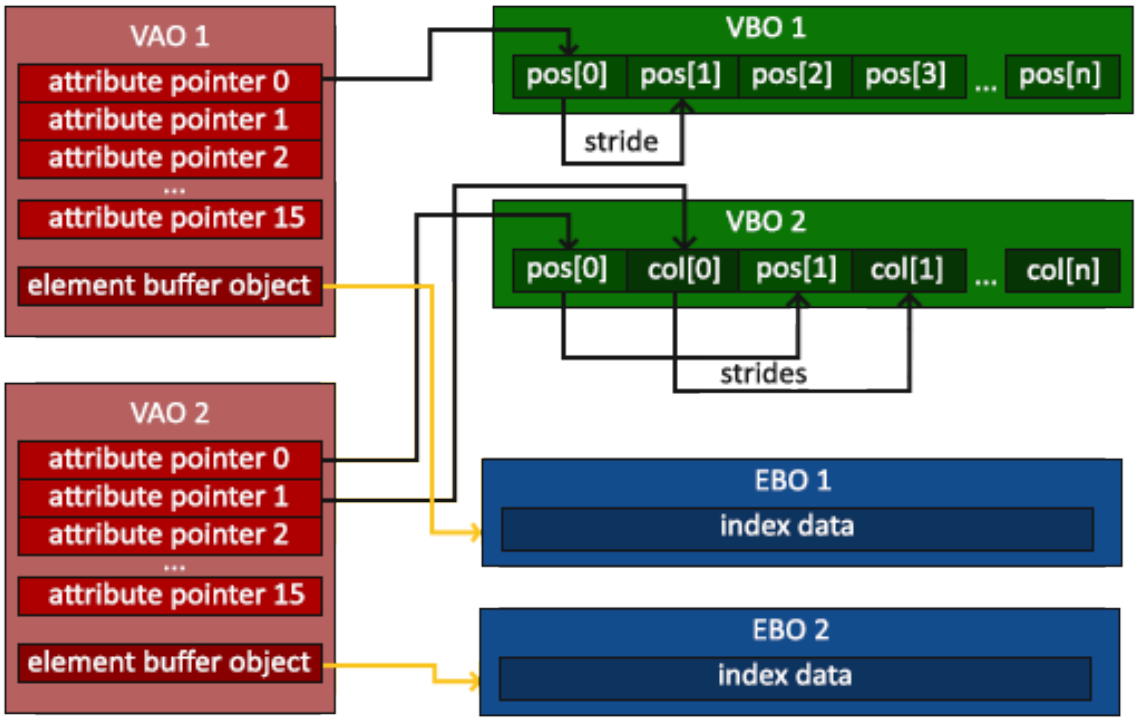
```
// El BackColorBuffer queda del color fijado
// con glClearColor()
// El DepthBuffer queda a 1 (máxima distancia)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// dibuja los objetos en el BackColorBuffer
scene.render();

// intercambia los buffers (Back/Front)
glfwSwapBuffers(mWindow);
```

Puntos

- Las primitivas operan sobre secuencias de vértices (`X`, `Y`, `Z`).
- Esta secuencia de vértices y sus atributos se suben a la GPU y se almacenan en *vertex buffer objects* (VBO) agregados en un *vertex array object* (VAO).



Puntos

- Los vértices se cargan una vez (`Mesh::load`) en la GPU.

```
glBindBuffer(GL_ARRAY_BUFFER, mVBO);
glBufferData(GL_ARRAY_BUFFER, vs.size() * sizeof(vec3),
             vs.data(), GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      sizeof(vec3), nullptr);
```

- Se utiliza las veces que sea para pintar la malla con la primitiva (`Mesh::render`).

```
glBindVertexArray(VaoId); // activa el VAO
glDrawArrays(GL_POINTS, 0, vs.size());
glBindVertexArray(0); // desactiva el VAO
```

Puntos

Ejemplo: Para definir las coordenadas de 4 vértices:

```
GLuint numVertices = 4;
std::vector<glm::vec3> vertices.reserve(numVertices);
vertices.emplace_back(10.0, 0.0, 0.0);
vertices.emplace_back(0.0, 10.0, 0);
vertices.emplace_back(0.0, 0.0, 10.0);
vertices.emplace_back(0.0, 0.0, 0.0);
```

Puntos: Atributos

- El grosor de las líneas será el que esté establecido en el momento de llamar a `glDrawArrays(...)`.
- Para dibujar todos los puntos con un grosor y color determinado: `glPointSize(GLfloat)`, `glColor*(...)`

```
glPointSize(3);          // obsoleto con size > 1
mesh->render();
glPointSize(1);          // valor por defecto
```

- En el perfil de compatibilidad, el color se fija con `glColor3d` y semejantes, pero aquí se fija a través del shader correspondiente.

Líneas

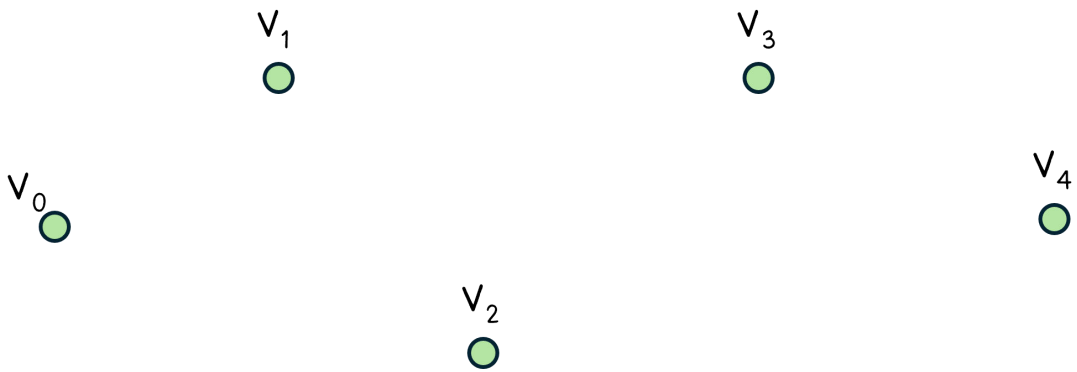
- Función: `glDrawArrays(mode, 0, numVertices);`
  👉 `mode` define la manera en la que se dibujan las líneas.

Líneas

`GL_LINES`

- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Cada par de vértices se interpreta como una línea v_0v_1, v_2v_3, ..., $v_{n-1}v_n$.
- Si el número de vértices es impar, el último vértice se ignora.

```
glDrawArrays(GL_LINES, 0, numVertices);
```

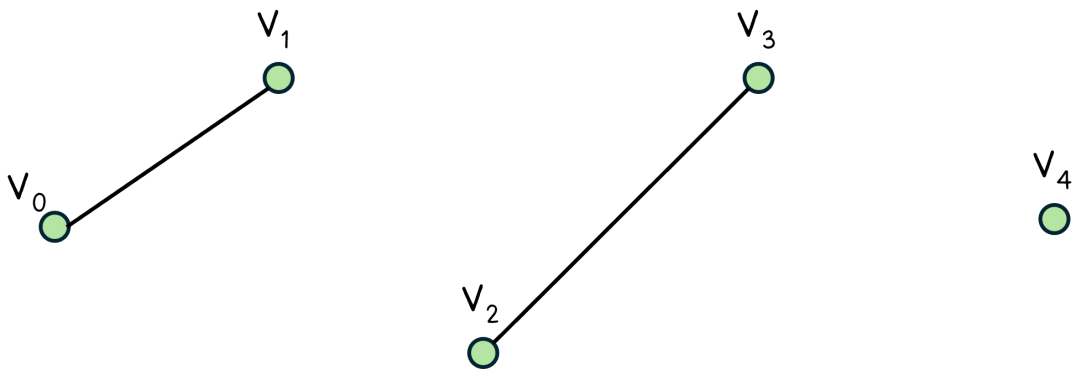


Líneas

GL_LINES

- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Cada par de vértices se interpreta como una línea v_0v_1, v_2v_3, ..., $v_{n-1}v_n$.
- Si el número de vértices es impar, el último vértice se ignora.

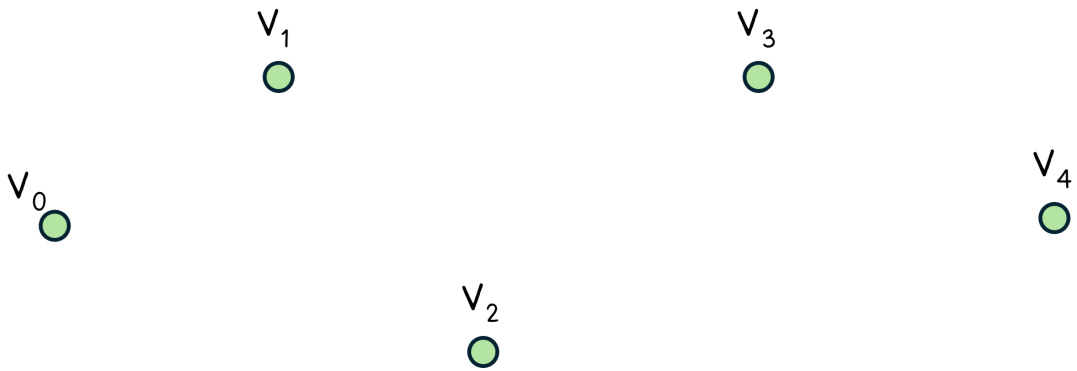
```
glDrawArrays(GL_LINES, 0, numVertices);
```



Líneas

GL_LINE_STRIP

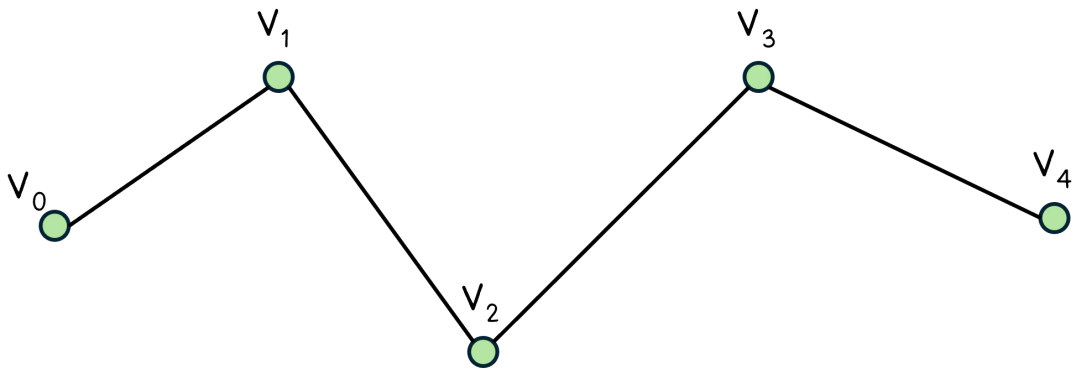
- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Se dibujan las líneas conectadas v_0v_1, v_1v_2, v_2v_3, ..., $v_{n-1}v_n$.
- Si el número de vértices es 1, no hace nada.



Líneas

GL_LINE_STRIP

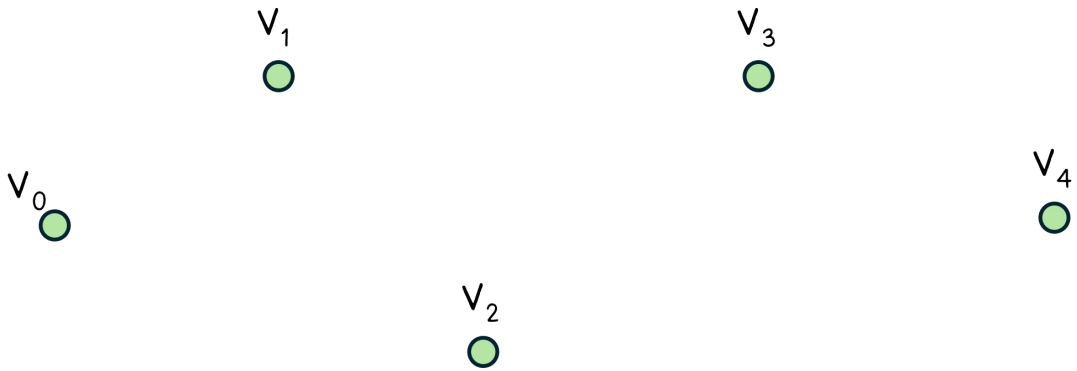
- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Se dibujan las líneas conectadas v_0v_1, v_1v_2, v_2v_3, ..., $v_{n-1}v_n$.
- Si el número de vértices es 1, no hace nada.



Líneas

GL_LINE_LOOP

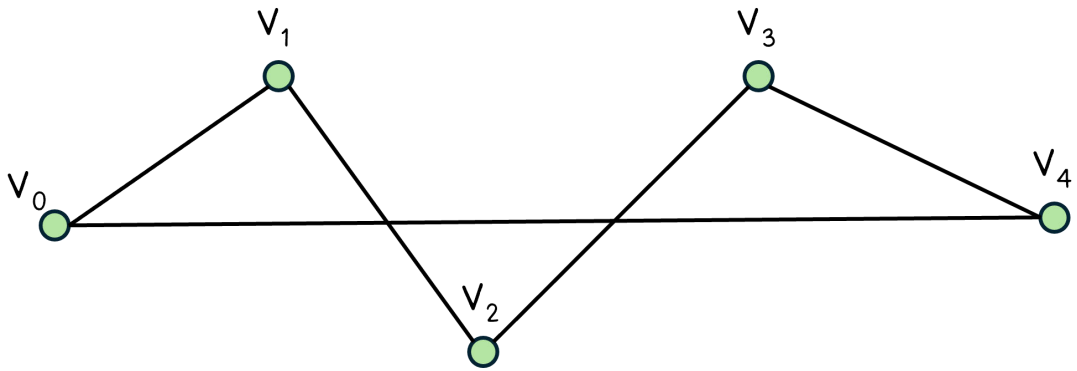
- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Se dibujan las líneas conectadas, con la unión entre el primer y el último vértice.
- Es decir, se dibujan las líneas v_0v_1, v_1v_2, ..., $v_{n-1}v_n$, v_nv_0.



Líneas

GL_LINE_LOOP

- Para vértices $v_0, v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- Se dibujan las líneas conectadas, con la unión entre el primer y el último vértice.
- Es decir, se dibujan las líneas v_0v_1, v_1v_2, ..., $v_{n-1}v_n$, v_nv_0.



Ejes RGB

```
generateAxesRGB(GLdouble l) {
  int numVertices = 6;
  std::vector<glm::vec3> vertices, colors;
  vertices.reserve(numVertices);
  vertices.emplace_back(0 ,0, 0);
  vertices.emplace_back(l, 0, 0);
  vertices.emplace_back(0, 0, 0);
  vertices.emplace_back(0, l, 0);
  vertices.emplace_back(0, 0, 0);
  vertices.emplace_back(0, 0, l);
  colors.reserve(numVertices);
  colors.emplace_back(1, 0, 0);
  colors.emplace_back(1, 0, 0);
  colors.emplace_back(0, 1, 0);
  colors.emplace_back(0, 1, 0);
  colors.emplace_back(0, 0, 1);
  colors.emplace_back(0, 0, 1); }
```

Orientación de los polígonos

- Un polígono tiene dos caras: una delantera y una trasera.
- Para identificar cuál es la delantera y cuál la trasera, basta con observar que, al mirar la cara delantera (o cara exterior), los vértices estarán dibujados en sentido **anti-horario** (**C**ounter-**C**lock **W**ise).

Modo

```
glPolygonMode(GL_FRONT_AND_BACK, GLenum mode);
```

- Especifica el modo en el cuál se rasterizará el polígono:
  - **mode** puede ser: `GL_FILL`, `GL_LINE` o `GL_POINT`.
  - antiguamente el primer argumento podía ser también `GL_FRONT` o `GL_BACK` separadamente.

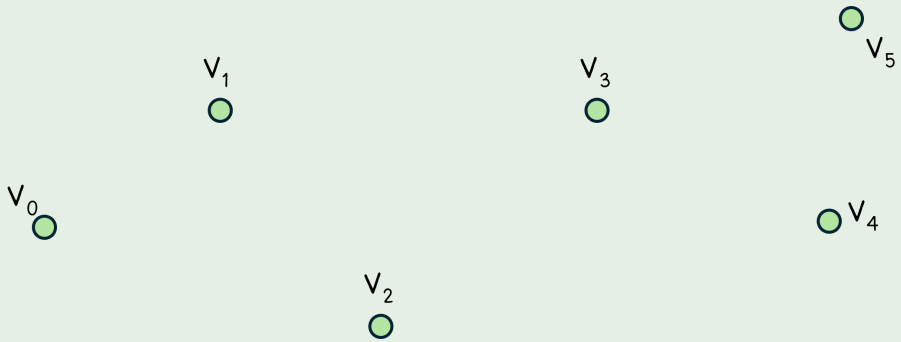
Triángulos

```
glDrawArrays(GL_TRIANGLES, 0, numVertices);
```

- Cada triplete de vértices se interpreta como un triángulo.

Ejemplo

Para vértices $v_0, v_1, v_2, v_3, v_4, v_5$, dibuja los triángulos independientes: $v_0v_1v_2$, $v_3v_4v_5$.



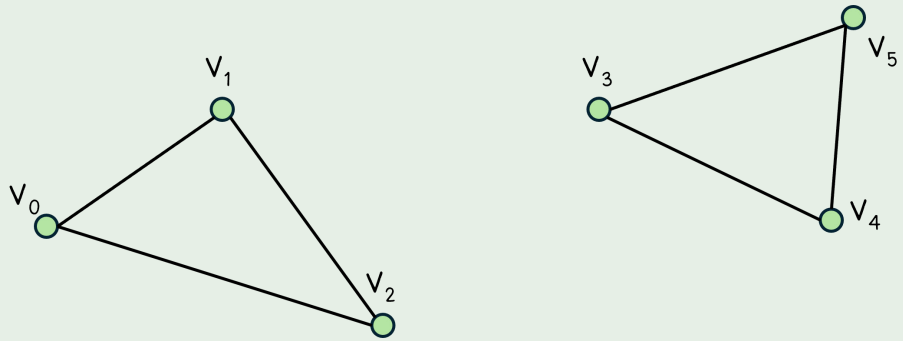
Triángulos

```
glDrawArrays(GL_TRIANGLES, 0, numVertices);
```

- Cada triplete de vértices se interpreta como un triángulo.

Ejemplo

Para vértices $v_0, v_1, v_2, v_3, v_4, v_5$, dibuja los triángulos independientes: $v_0v_1v_2$, $v_3v_4v_5$.



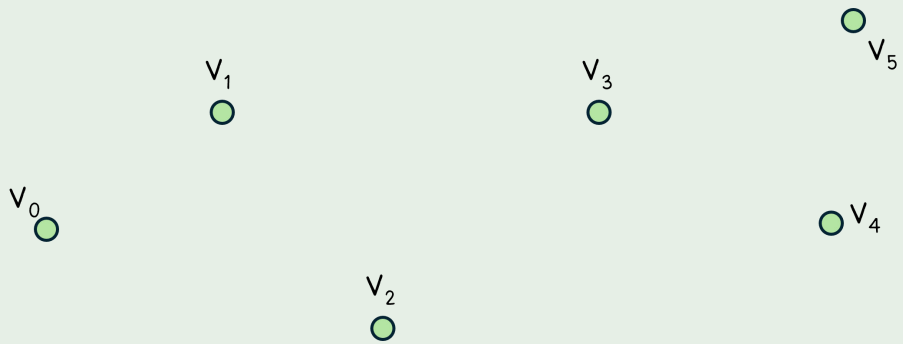
Triángulos

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, numVertices);
```

- Se dibuja un triángulo, y cada nuevo vértice se interpreta con un triángulo entre los dos anteriores vértices y el nuevo.
- El número de vértices tiene que ser al menos 3.

Ejemplo

Para vértices $v_0, v_1, v_2, v_3, v_4, v_5, v_6$, dibuja los triángulos: $v_0v_1v_2$, $v_1v_2v_3$, $v_2v_3v_4$, $v_3v_4v_5$, $v_4v_5v_6$ uniformizando el sentido CCW con el del primer triángulo. Por tanto, dibuja los triángulos: $v_0v_1v_2$, $v_2v_1v_3$, $v_2v_3v_4$, $v_4v_3v_5$, $v_4v_5v_6$.



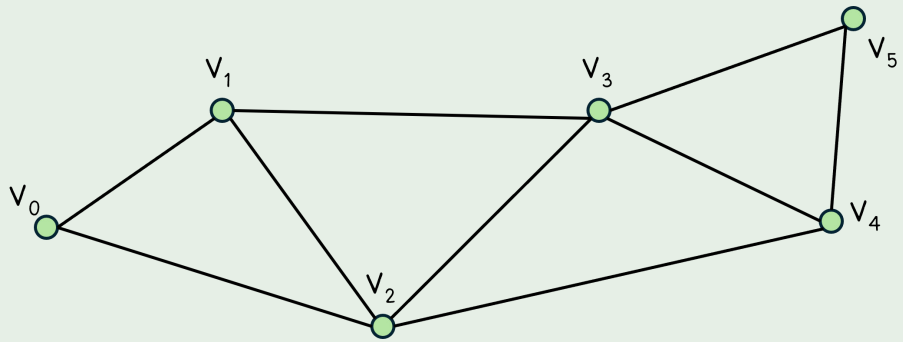
Triángulos

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, numVertices);
```

- Se dibuja un triángulo, y cada nuevo vértice se interpreta con un triángulo entre los dos anteriores vértices y el nuevo.
- El número de vértices tiene que ser al menos 3.

Ejemplo

Para vértices $v_0, v_1, v_2, v_3, v_4, v_5, v_6$, dibuja los triángulos: $v_0v_1v_2$, $v_1v_2v_3$, $v_2v_3v_4$, $v_3v_4v_5$, $v_4v_5v_6$ uniformizando el sentido CCW con el del primer triángulo. Por tanto, dibuja los triángulos: $v_0v_1v_2$, $v_2v_1v_3$, $v_2v_3v_4$, $v_4v_3v_5$, $v_4v_5v_6$.



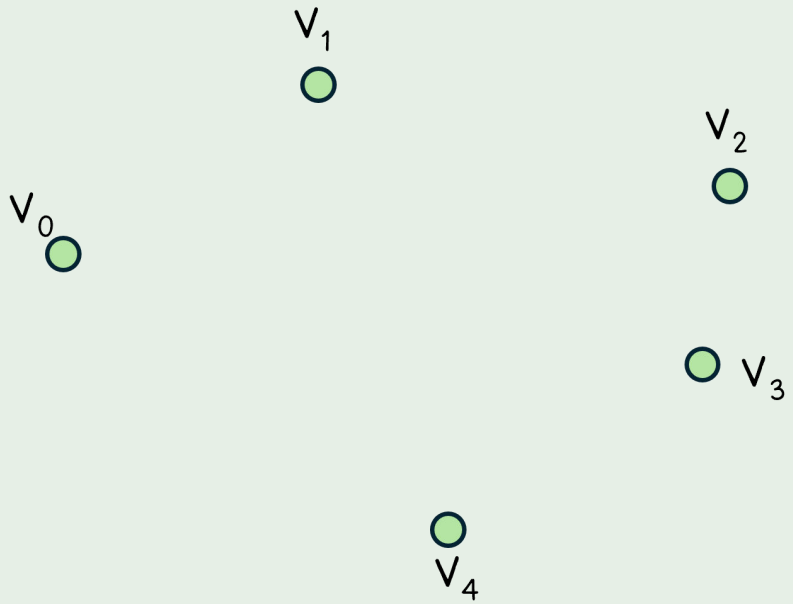
Triángulos

```
glDrawArrays(GL_TRIANGLE_FAN, 0, numVertices);
```

- Se dibujan triángulos con un vértice común.

Ejemplo

Para vértices v_0, v_1, v_2, v_3, v_4, dibuja los triángulos: $v_0v_1v_2$, $v_0v_2v_3$, $v_0v_3v_4$. Todos los triángulos comparten un vértice común: v_0.



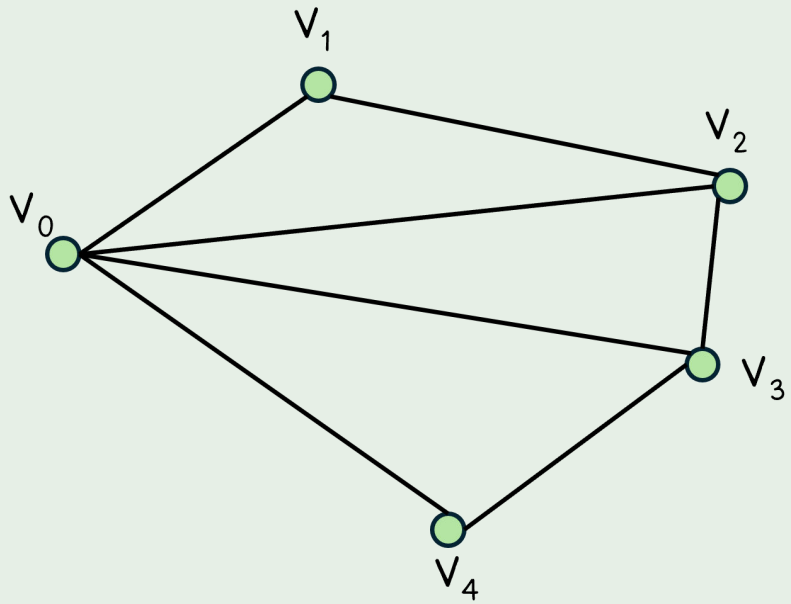
Triángulos

```
glDrawArrays(GL_TRIANGLE_FAN, 0, numVertices);
```

- Se dibujan triángulos con un vértice común.

Ejemplo

Para vértices v_0, v_1, v_2, v_3, v_4, dibuja los triángulos: $v_0v_1v_2$, $v_0v_2v_3$, $v_0v_3v_4$. Todos los triángulos comparten un vértice común: v_0.

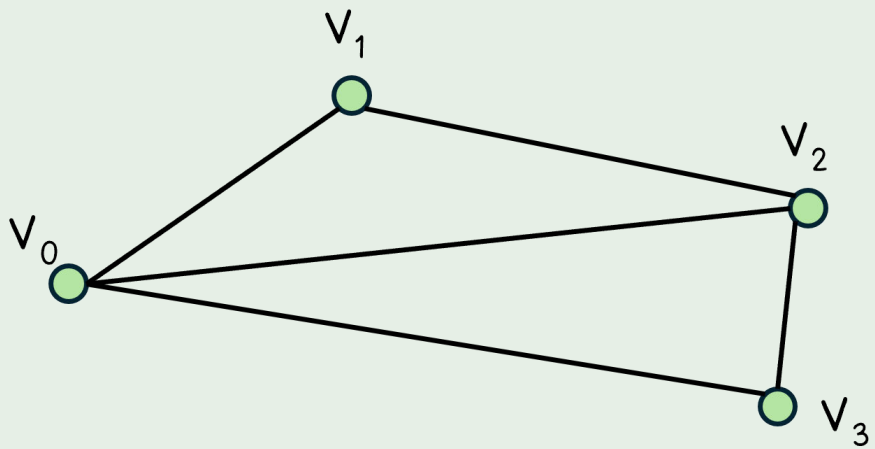


Cuadriláteros

- Para cuadriláteros, se utiliza `GL_TRIANGLE_STRIP`.

Ejemplo

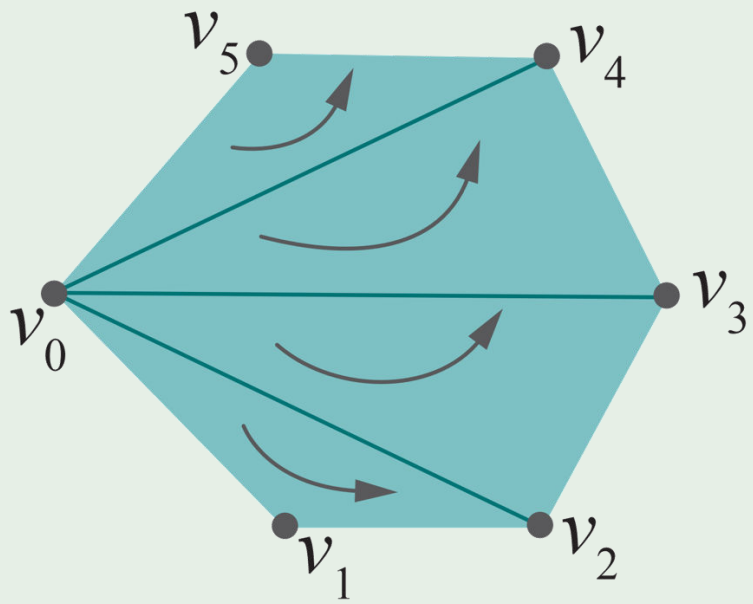
Para los cuatro vértices del cuadrilátero $v_0v_1v_2v_3$, dados en el orden $v_0v_1v_2v_3$, dibuja el cuadrilátero con 2 triángulos: $v_0v_1v_2$ y $v_2v_1v_3$.



Polígonos

- Para polígonos, se utiliza `GL_TRIANGLE_FAN` con los vértices del polígono $v_0 v_1 v_2 v_3 \ldots v_n$ en orden contrario a las agujas del reloj.

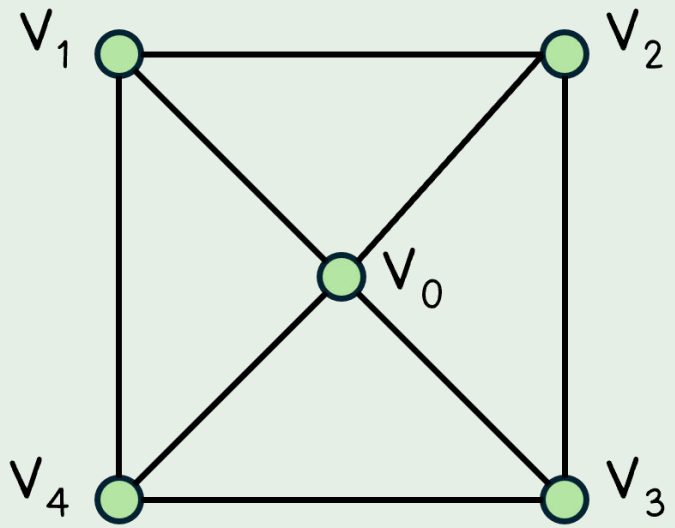
Ejemplo



Polígonos

- Para polígonos, se utiliza `GL_TRIANGLE_FAN` con los vértices del polígono $v_0 v_1 v_2 v_3 \ldots v_n$ en orden contrario a las agujas del reloj.

Ejemplo



V_0, V_1, V_2, V_3, V_4

Transformaciones con GLM

Existen 3 tipos de transformaciones afines:

- Traslación: cambian la **posición** de un objeto.
- Rotación: cambia la **orientación** de un objeto, sin deformar el objeto (transformaciones rígidas).
- Escala: las escalas uniformes cambian el **tamaño** de un objeto. Las escales no uniformes pueden deformar el objeto.

Traslaciones con GLM

```
glm::mat4 m = glm::translate(mat4, vec3);
```

- `translate(mat4(1), vec3(tx, ty, tz))` es la matriz de translación

$$M_T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Composición: `m = translate(m0, vec3(tx, ty, tz));`

$$M = M_0 \cdot M_T$$

$$M \cdot V = (M_0 \cdot M_T) \cdot V = M_0 \cdot (M_T \cdot V)$$

Escalas con GLM

```
glm::mat4 m = glm::scale(mat4, vec3);
```

- `scale(mat4(1), vec3(sx, sy, sz))` es la matriz de escala:

$$M_S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Composición: `m = scale(m0, vec3(sx, sy, sz));`

$$M = M_0 \cdot M_S$$

$$M \cdot V = (M_0 \cdot M_S) \cdot V = M_0 \cdot (M_S \cdot V)$$

Rotaciones con GLM

`glm::mat4 m = glm::rotate(mat4, β , vec3);`

- Para **Z-Roll**, `rotate(mat4(1), β, vec3(0, 0, 1))` es la matriz de rotación

$$M_R = \begin{pmatrix} cos(\beta) & -sin(\beta) & 0 & 0 \\ sin(\beta) & cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotaciones con GLM

`glm::mat4 m = glm::rotate(mat4, β , vec3);`

- Para **Y-Yaw**, `rotate(mat4(1), β, vec3(0, 1, 0))` es la matriz de rotación

$$M_R = \begin{pmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotaciones con GLM

`glm::mat4 m = glm::rotate(mat4, β , vec3);`

- Para **X-Pitch**, `rotate(mat4(1), β, vec3(1, 0, 0))` es la matriz de rotación

$$M_R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\beta) & -sin(\beta) & 0 \\ 0 & sin(\beta) & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Composición de transformaciones en OpenGL

- En OpenGL (GLM) las transformaciones se componen post-multiplicando las matrices: `M = transformar(M_0, ...);` $\rightarrow M = M_0 \cdot M_A$
- Al aplicar `M` a un vértice: $M \cdot V = (M_0 \cdot M_A) \cdot V = M_0 \cdot (M_A \cdot V)$

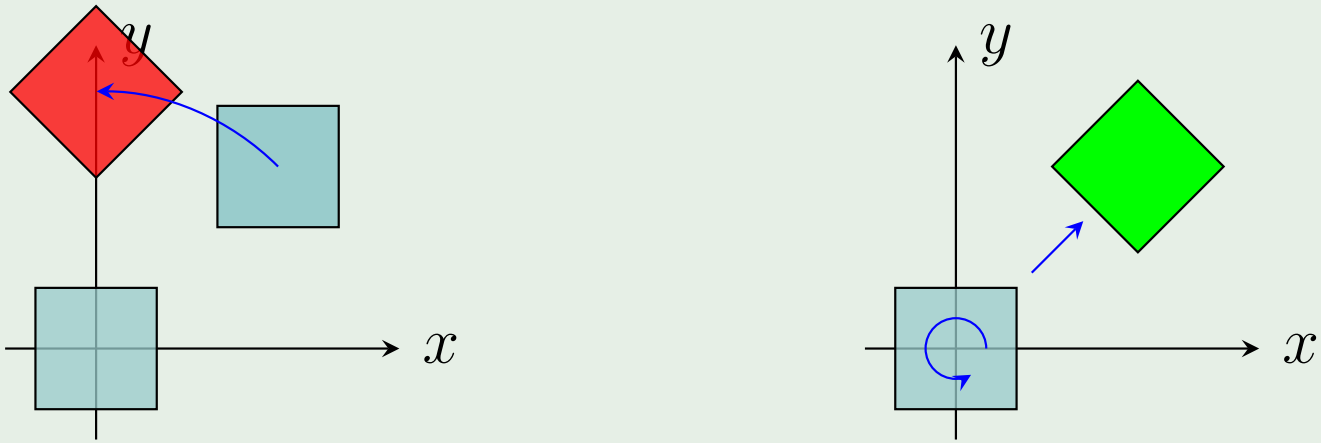
Composición de transformaciones en OpenGL

Ejemplo

Tenemos un cuadrado centrado y alineado con los ejes, y queremos situarlo en el punto $(7{,}5, 7{,}5, 0)$ girado 45° sobre su centro.

```
m = rotate(mI, radians(45.0), vec3(0,0,1));
m = translate(m, vec3(7.5,7.5,0));
```

$\rightarrow m = mI \times mR \times mT$



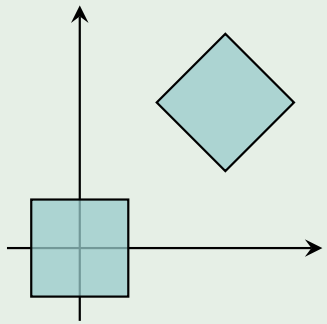
Composición de transformaciones en OpenGL

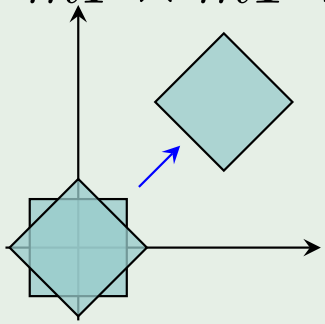
Ejemplo

Tenemos un cuadrado centrado y alineado con los ejes, y queremos situarlo en el punto $(7{,}5, 7{,}5, 0)$ girado 45° sobre su centro.

```
m = translate(mI, vec3(7.5,7.5,0));
m = rotate(m, radians(45.0), vec3(0,0,1));
```

$\rightarrow m = mI \times mT \times mR$





```
mT = translate(mI, vec3(7.5,7.5,0));
mR = rotate(mI, radians(45.0), vec3(0,0,1));
```

$m = mT \times mR$

Composición de transformaciones

- **Animación de objetos**: es habitual trabajar con una ruta por la que se **desplaza** el objeto (`mT`) y la **orientación** del objeto (`mR`) mientras se desplaza. Se sitúa y orienta al objeto desde sus coordenadas originales.
  Estando el objeto centrado en coordenadas locales:

$$\texttt{matriz de modelado del objeto} = mT \times mR$$

  Si es necesario escalar el objeto (`mS`):

$$\texttt{matriz de modelado del objeto} = mT \times mR \times mS$$

Composición de transformaciones

- Y por último la matriz de vista:

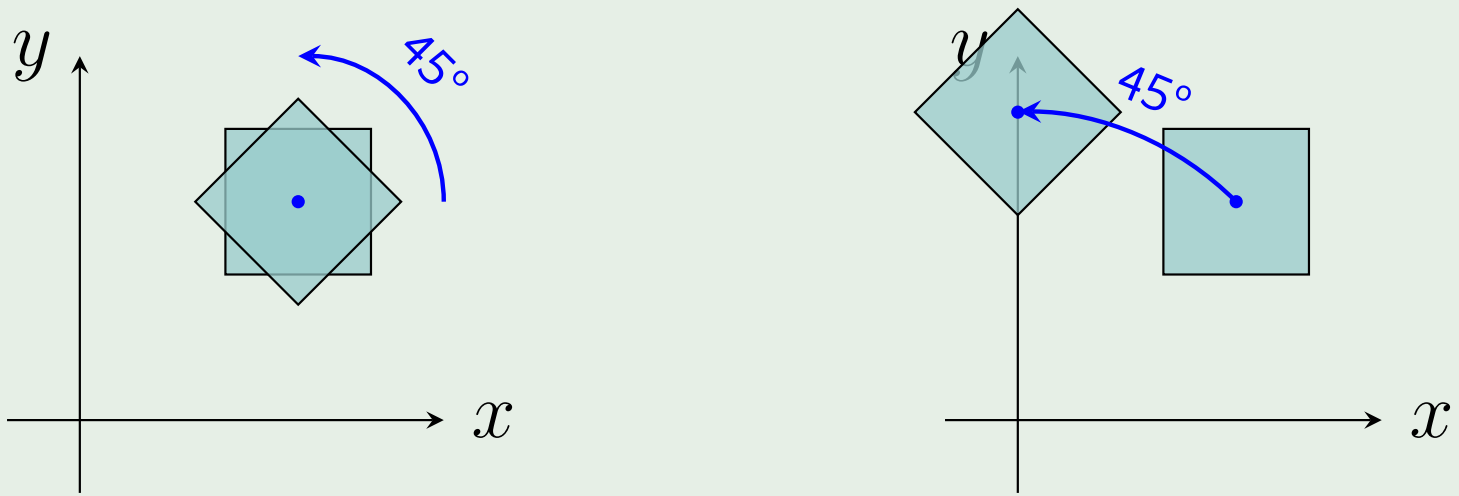
$$\texttt{matriz de modelado y vista} = \texttt{matriz de vista} \times \texttt{matriz de modelado}$$

- **La matriz de vista es la inversa de la matriz de modelado de la cámara**: en lugar de colocar la cámara en la escena (matriz de la cámara) coloca los objetos de la escena con respecto a la cámara.

Composición de transformaciones

Ejemplo

Tenemos un cuadrado alineado con los ejes con centro en $(7{,}5, 7{,}5, 0{,}0)$. Queremos rotarlo 45° sobre su centro.



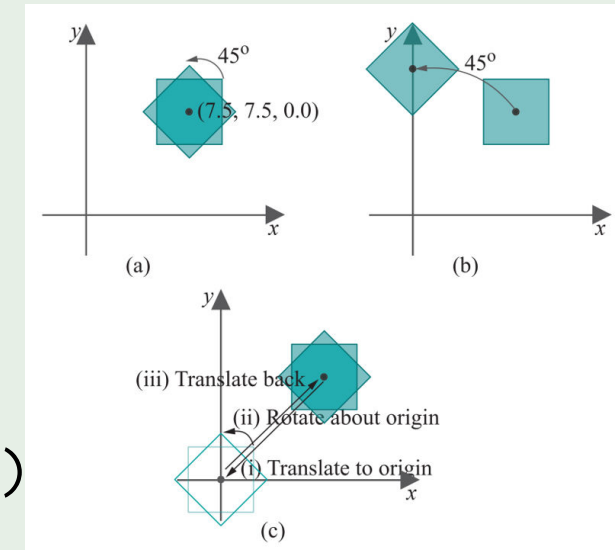
Composición de transformaciones

Ejemplo

Tenemos un cuadrado alineado con los ejes con centro en $(7{,}5, 7{,}5, 0{,}0)$. Queremos rotarlo 45° sobre su centro.

```
m = translate(mI, vec3(7.5,7.5,0));
m = rotate(m, radians(45.0), vec3(0,0,1));
m = translate(m, vec3(-7.5,-7.5,0));
```

$\rightarrow m = mI \times mT \times mR \times mT{-}1$




```
mT = translate(mI, vec3(7.5,7.5,0));
mR = rotate(mI, radians(45.0), vec3(0,0,1))
mTi = translate(mI, vec3(-7.5,-7.5,0));
```

$m = mT \times mR \times mT_i$

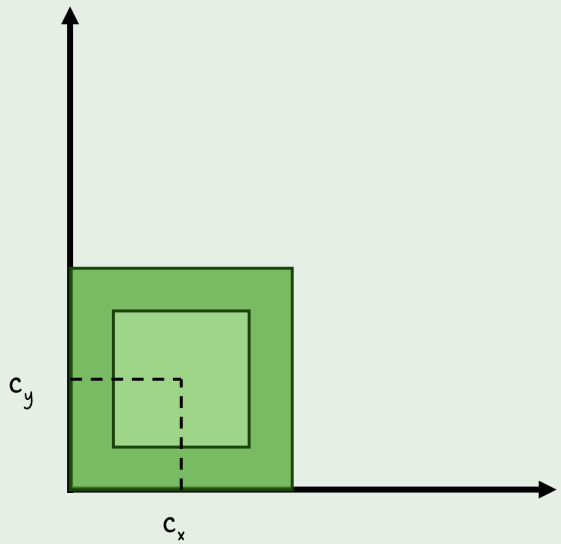
Composición de transformaciones

Ejemplo

Tenemos un cuadrado alineado con los ejes con centro en $(c_x, c_y, 0, 0)$. Queremos escalarlo sobre su centro (sin modificar el centro).

```
m = translate(mI, vec3(cx,cy,0));
m = scale(m, vec3(2,2,2));
m = translate(m, vec3(-cx,-cy,0));
```

$\rightarrow M = M_I \cdot M_T \cdot M_S \cdot M_{T-1}$



CUIDADO!!! MAL

```
m = scale(mI, vec3(2,2,2));
```

$\rightarrow m = mI \times mS$

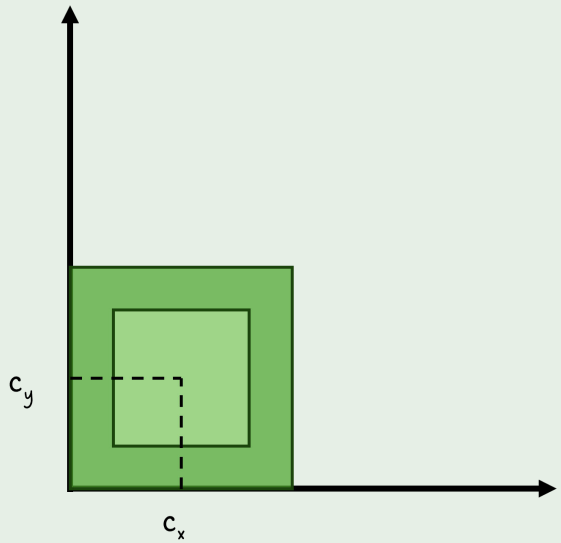
Composición de transformaciones

Ejemplo

Tenemos un cuadrado alineado con los ejes con centro en $(c_x, c_y, 0, 0)$. Queremos escalarlo sobre su centro (sin modificar el centro).

```
mT = translate(mI, vec3(cx,cy,0));
mS = scale(mI, vec3(2,2,2));
mTi = translate(mI, vec3(-cx,-cy,0));
```

$$M = M_T \times M_S \times M_{T_i};$$



Transformaciones

$$\texttt{ModelView MATRIX} = \text{VIEW MATRIX} \times \text{MODEL MATRIX}$$