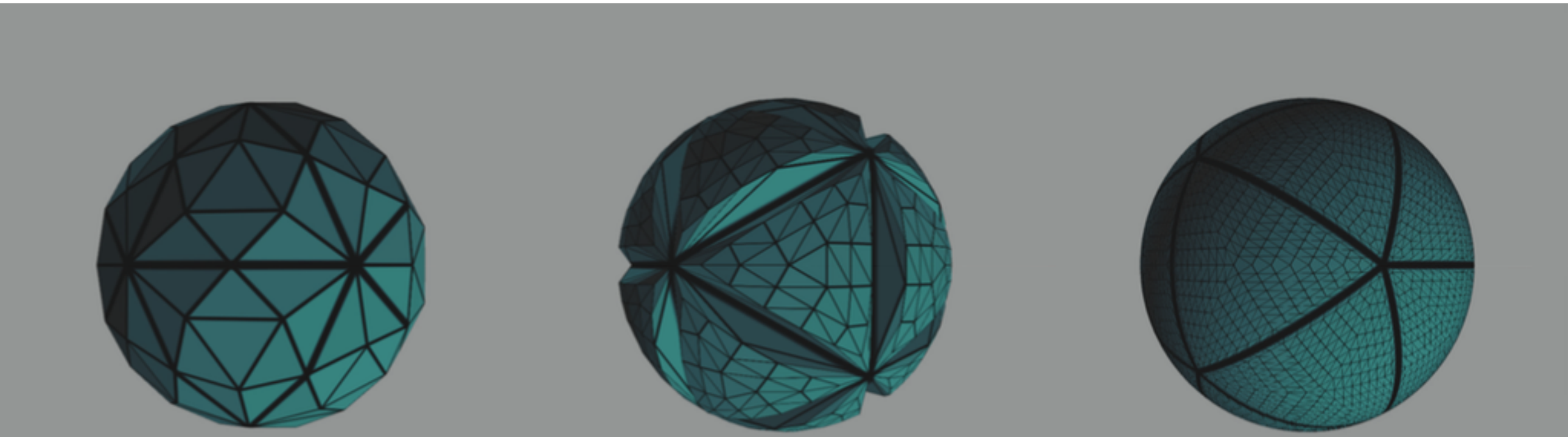
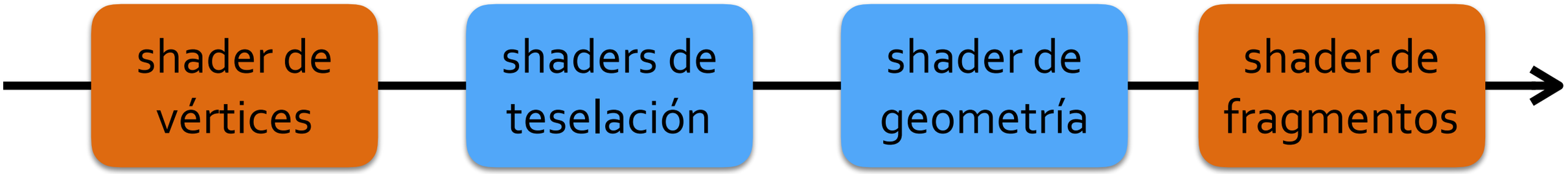
OpenGL en Processing y shaders de geometría y teselación



El pipeline gráfico



No soportados en Processing :-(

…pero podemos accederlos desde OpenGL :-)

OpenGL en Processing


```
public void draw() {
  PGL pgl = beginPGL();

  background(0);

  // The geometric transformations will be automatically passed
  // to the shader.
  rotate(frameCount * 0.01f, width, height, 0);

  updateGeometry();
  sh.bind();

  // get "vertex" attribute location in the shader
  final int vertLoc = pgl.getAttribLocation(sh.glProgram, "vertex");
  // enable array for "vertex" attribute
  pgl.enableVertexAttribArray(vertLoc);

  // get "color" attribute location in the shader
  final int colorLoc = pgl.getAttribLocation(sh.glProgram, "color");
  // enable array for "color" attribute
  pgl.enableVertexAttribArray(colorLoc);

  // ...

  endPGL();
}
```

Versiones de OpenGL y GLSL

Versión	Años	Características	Versión de GLSL
1.1 - 1.5	1997 - 2003	Pipeline fijo	-
2.0, 2.1	2004, 2006	Shaders (opcionales)	1.10, 1.20
3.0 - 3.3	2008-2009	Shaders (obligatorios), Shaders de geometría	1.30 - 3.30
4.0 - 4.5	2010 - 2014	Shaders de teselación y de computación	4.00 - 4.50

OpenGL ES (for Embedded Systems)

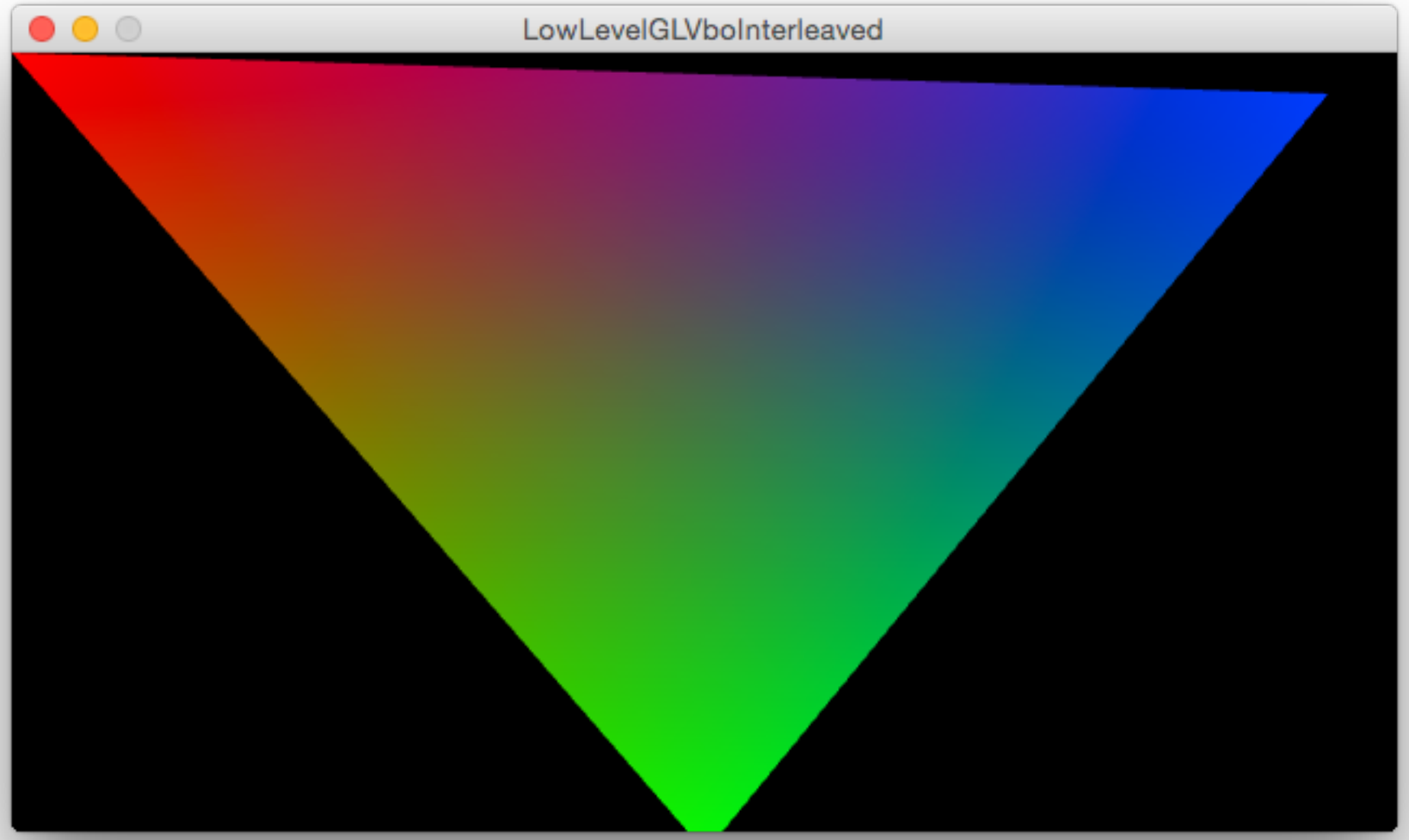
Es (más o menos) un subconjunto de OpenGL

Versión	Año de introducción	Características	Subconjunto de OpenGL…
1.x	2003	Pipeline fijo	1.x
2.0	2007	Shaders (obligatorios)	2.x ~ 3.x
3.x	2012	Shaders de geometría, teselación y computación	4.x

¿Que usa Processing?

- Processing expone una interface GLES 2 con su clase PGL
- Esto nos permite programar OpenGL de manera de asegurarnos que sea compatible entre desktop y Android
- Los shaders en Processing se escriben por defecto en GLSL 1.10, que es compatible entre desktop y Android (y tambien con WebGL 1.0)

¡Un triángulo en OpenGL!



```
void draw() {
  PGL pgl = beginPGL();
  background(0);
  sh.bind();

  int posLoc = pgl.getAttribLocation(sh.glProgram, "position");
  pgl.enableVertexAttribArray(posLoc);
  int colorLoc = pgl.getAttribLocation(sh.glProgram, "color");
  pgl.enableVertexAttribArray(colorLoc);

  pgl.bindBuffer(PGL.ARRAY_BUFFER, vboId);
  int size = 4;
  int stride = (4 + 4) * Float.BYTES;
  int offset = 4 * Float.BYTES;
  pgl.vertexAttribPointer(posLoc, size, PGL.FLOAT, false, stride, 0);
  pgl.vertexAttribPointer(colorLoc, size, PGL.FLOAT, false, stride, offset);

  pgl.drawArrays(PGL.TRIANGLES, 0, 3);

  pgl.bindBuffer(PGL.ARRAY_BUFFER, 0);

  pgl.disableVertexAttribArray(posLoc);
  pgl.disableVertexAttribArray(colorLoc);

  sh.unbind();

  endPGL();
}
```

Referencia rápida de OpenGL ES y GLSL

OpenGL ES 2.0 API Quick Reference Card

OpenGL® ES is a software interface to graphics hardware. The interface consists of a set of procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

- [n.n.n] refers to sections and tables in the OpenGL ES 2.0 specification.
- [n.n.n] refers to sections in the OpenGL ES Shading Language 1.0 specification.

Specifications are available at www.opengl.org/registry/gles

Errors [2.5]

enum **GetError**(void); //Returns one of the following:

INVALID_ENUM	Enum argument out of range
INVALID_FRAMEBUFFER_OPERATION	Framebuffer is incomplete
INVALID_VALUE	Numeric argument out of range
INVALID_OPERATION	Operation illegal in current state
OUT_OF_MEMORY	Not enough memory left to execute command
NO_ERROR	No error encountered

OpenGL ES Command Syntax [2.3]

Open GL ES commands are formed from a return type, a name, and optionally a type letter i for 32-bit int, or f for 32-bit float, as shown by the prototype below:

return-type **Name**{1234}{if}{v} (*[args ,] T arg1 , . . . , T argN [, args]*);

The arguments enclosed in brackets ([args ,] and [, args]) may or may not be present.
The argument type T and the number *N* of arguments may be indicated by the command name suffixes. *N* is 1, 2, 3, or 4 if present, or else corresponds to the type letters. If "v" is present, an array of *N* items is passed by a pointer.

For brevity, the OpenGL documentation and this reference may omit the standard prefixes.
The actual names are of the forms: **glFunctionName()**, **GL_CONSTANT**, **GLtype**

GL Data Types [2.3]

GL types are not C types.

GL Type	Minimum Bit Width	Description
boolean	1	Boolean
byte	8	Signed binary integer
ubyte	8	Unsigned binary integer
char	8	Characters making up strings
short	16	Signed 2's complement binary integer
ushort	16	Unsigned binary integer
int	32	Signed 2's complement binary integer
uint	32	Unsigned binary integer
fixed	32	Signed 2's complement 16.16 scaled integer
sizei	32	Non-negative binary integer size
enum	32	Enumerated binary integer value
intptr	*ptrbits*	Signed 2's complement binary integer
sizeiptr	*ptrbits*	Non-negative binary integer size
bitfield	32	Bit field
float	32	Floating-point value
clampf	32	Floating-point value clamped to [0; 1]

Buffer Objects [2.9]

Buffer objects hold vertex array data or indices in high-performance server memory.

void **GenBuffers**(sizei *n*, uint **buffers*);
void **DeleteBuffers**(sizei *n*, const uint **buffers*);

Creating and Binding Buffer Objects

void **BindBuffer**(enum *target*, uint *buffer*);
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER

Creating Buffer Object Data Stores

void **BufferData**(enum *target*, sizeiptr *size*, const void **data*, enum *usage*);
usage: STATIC_DRAW, STREAM_DRAW, DYNAMIC_DRAW

Updating Buffer Object Data Stores

void **BufferSubData**(enum *target*, intptr *offset*, sizeiptr *size*, const void **data*);
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER

Buffer Object Queries [6.1.6, 6.1.3]

boolean **IsBuffer**(uint *buffer*);
void **GetBufferParameteriv**(enum *target*, enum *value*, T *data*);
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER
value: BUFFER_SIZE, BUFFER_USAGE

Viewport and Clipping

Controlling the Viewport [2.12.1]

void **DepthRangef**(clampf *n*, clampf *f*);
void **Viewport**(int *x*, int *y*, sizei *w*, sizei *h*);

Reading Pixels [4.3.1]

void **ReadPixels**(int *x*, int *y*, sizei *width*, sizei *height*, enum *format*, enum *type*, void **data*);
format: RGBA type: UNSIGNED_BYTE
Note: ReadPixels() also accepts a queriable implementation-defined *format/type* combination, see [4.3.1].

Vertices

Current Vertex State [2.7]

void **VertexAttrib{1234}{f}**(uint *index*, T *values*);
void **VertexAttrib{1234}{f}v**(uint *index*, T *values*);

Vertex Arrays [2.8]

Vertex data may be sourced from arrays that are stored in application

TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}
format and *type:* See TexImage2D

Compressed Texture Images [3.7.3]

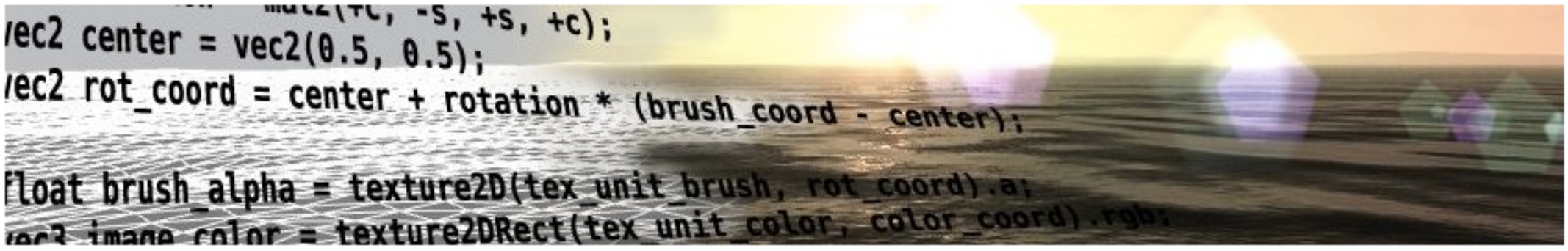
index: [0, MAX_VERTEX_ATTRIBS - 1]

void **DrawArrays**(enum *mode*, int *first*, sizei *count*);
void **DrawElements**(enum *mode*, sizei *count*, enum *type*,

Texture Image Specification [3.7.1]

void **TexImage2D**(enum *target*, int *level*, int *internalformat*, sizei *width*, sizei *height*, int *border*, enum *format*,

GLGraphics y GSVideo (2008 - 2011)



GLGraphics

GLGraphics

A library by Andres Colubri for the programming environment processing.
Last update, 10/04/2011.

GLGraphics is a library intended to extend the capabilities of the OPENGL renderer in Processing.
It includes classes to handle opengl textures, image post-processing filters, 3D Models, and shaders in GLSL, Cg and CgFX. It also includes an offscreen rendering surface with antialias support. Look at the examples below for the details on how to use GLGraphics.
Check codeanticode's blog and twitter feed for updates.
For usage feedback, please post to the Processing's contributed library forum.
For development support (bugs reports, feature requests), use sourcerfoge's trackers and forums.

Keywords. opengl, filters, effects, shaders, gpu

Reference. Have a look at the javadoc reference here. a copy of the reference is included in the .zip as well.

Source. The source code of GLGraphics is available at sourceforge.net, and its repository can be browsed here.

GSVideo

GSVideo

A library by Andres Colubri for the programming environment processing.
Last update, 10/04/2011.

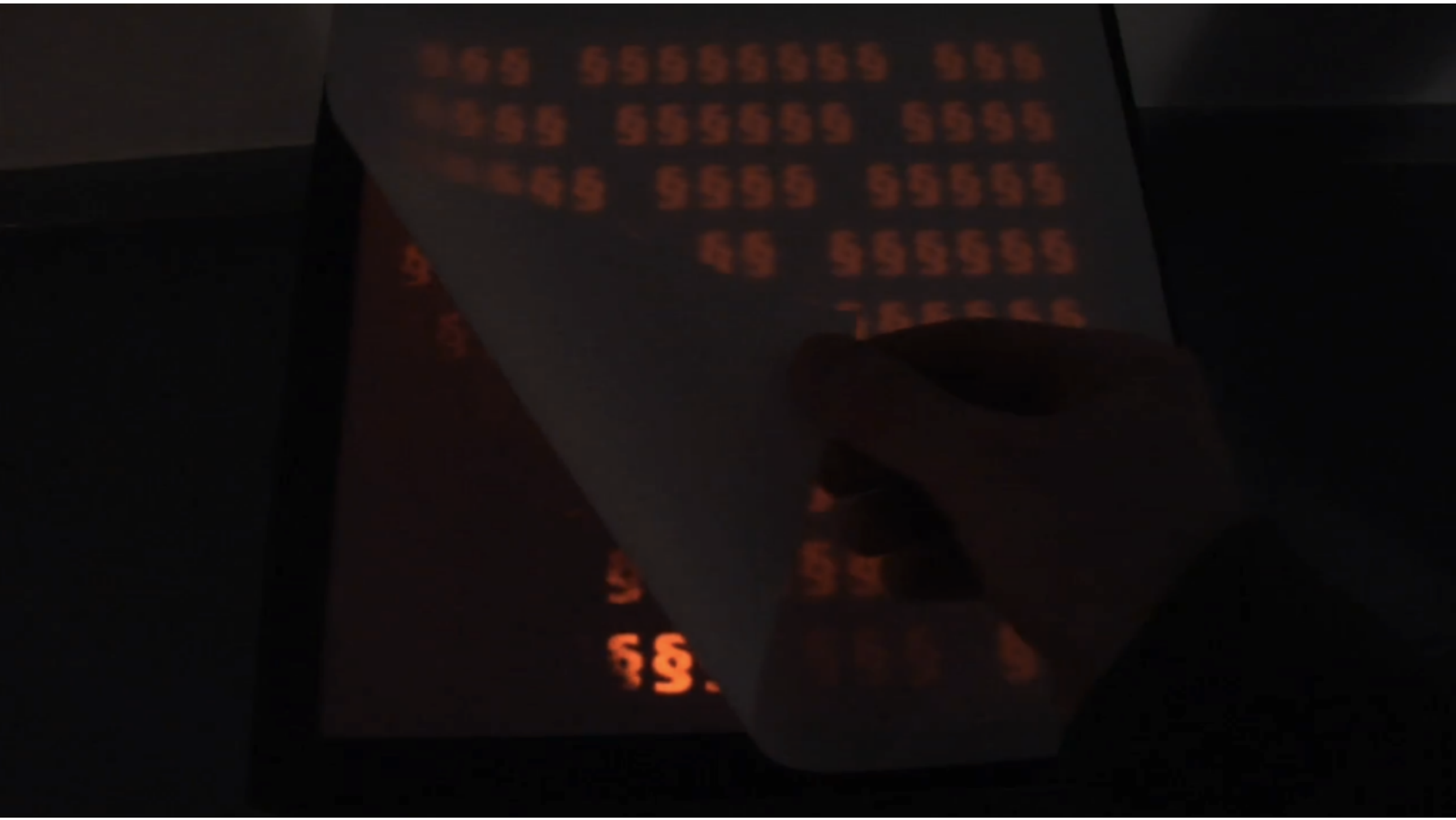
GSVideo is a library that offers video playback, capture and recording functionalities through the use of the GStreamer multimedia framework. It follows the API of the built-in video library. It also allows to create custom gstreamer pipelines inside Processing.
Check codeanticode's blog and twitter feed for updates.
For usage feedback, please post to the Processing's contributed library forum.
For development support (bugs reports, feature requests), use sourcerfoge's trackers and forums.

Keywords. video, playback, capture, gstreamer, crossplatform

Reference. Have a look at the javadoc reference here. a copy of the reference is included in the .zip as well.

Source. The source code of GSVideo is available at sourceforge.net, and its repository can be browsed here.

A_H1N1, por Konstantin Flick



https://player.vimeo.com/video/9452586

Interactive Spine, por Design & Systems

https://player.vimeo.com/video/55257216

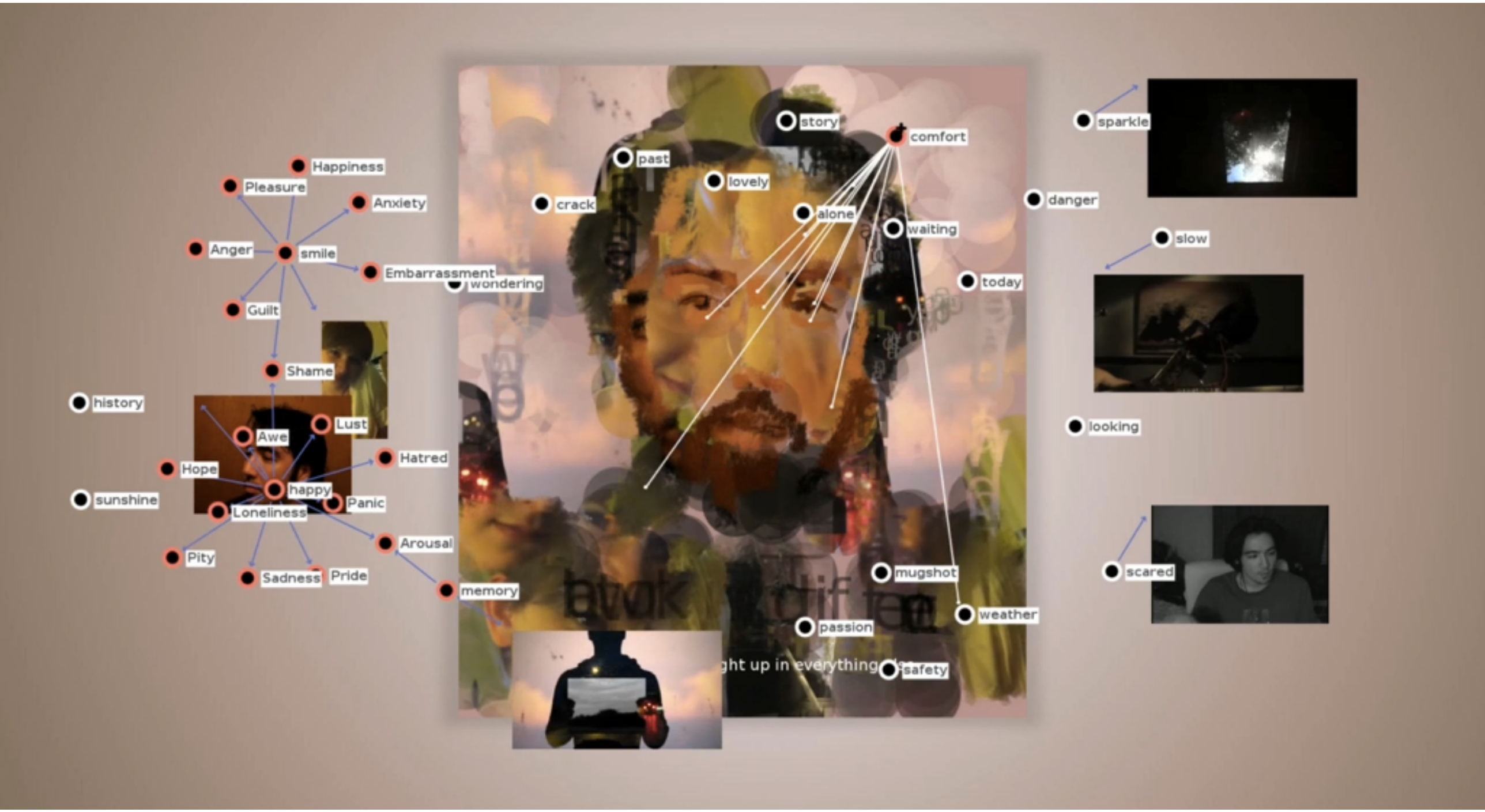
Generating Utopia, por Stefan Wagner

https://player.vimeo.com/video/74066023

Generative Typography, por Amnon Owed

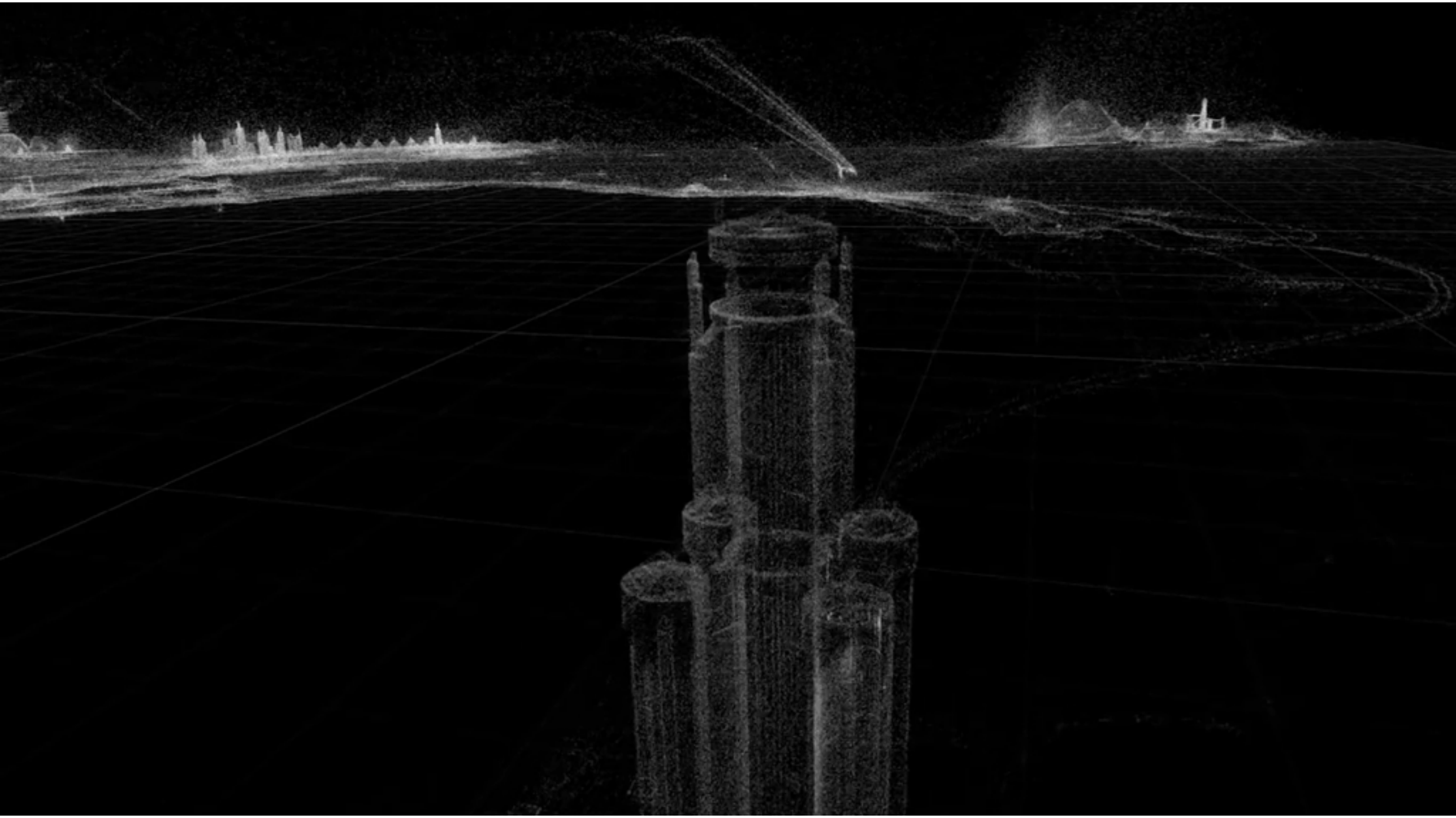
https://player.vimeo.com/video/101383026

Video portraits Sergio Albiac



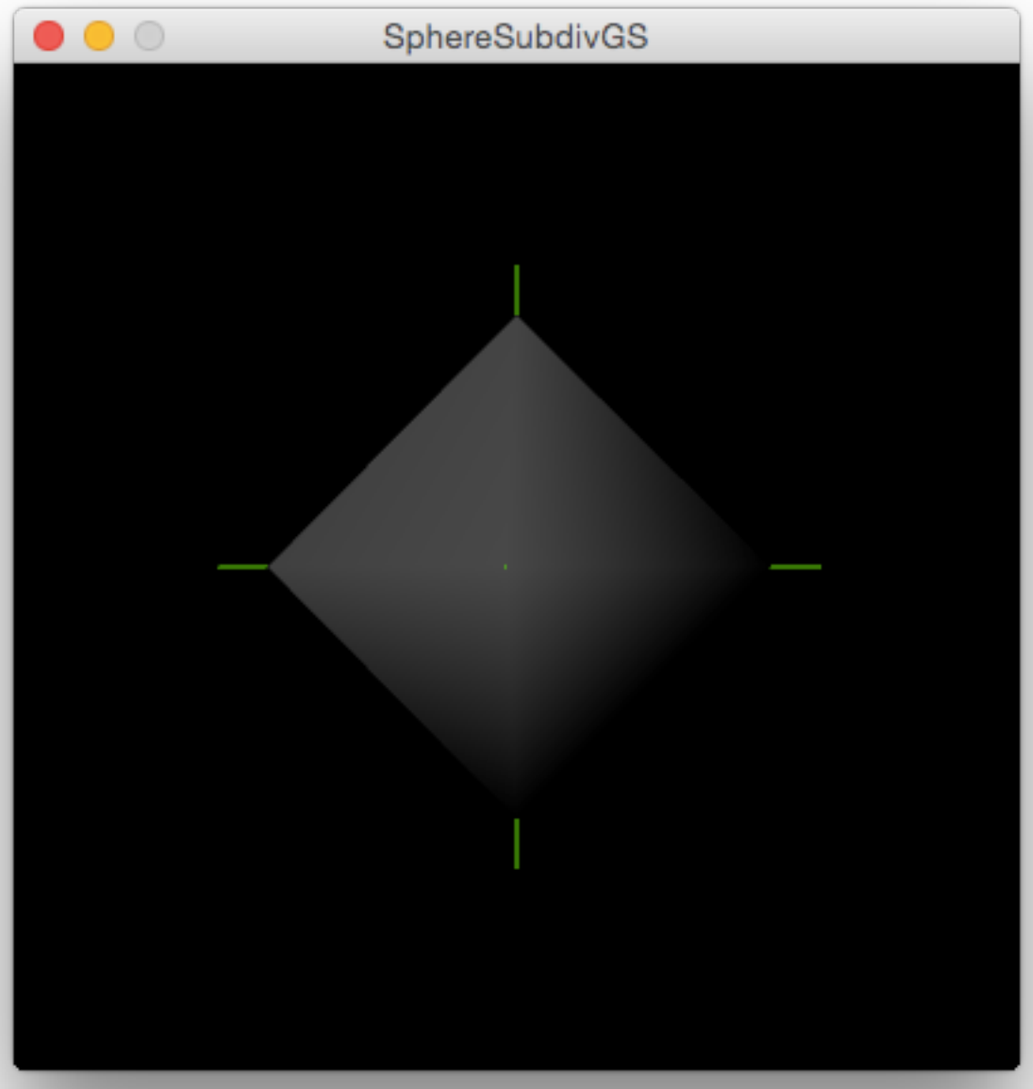
https://player.vimeo.com/video/32760578

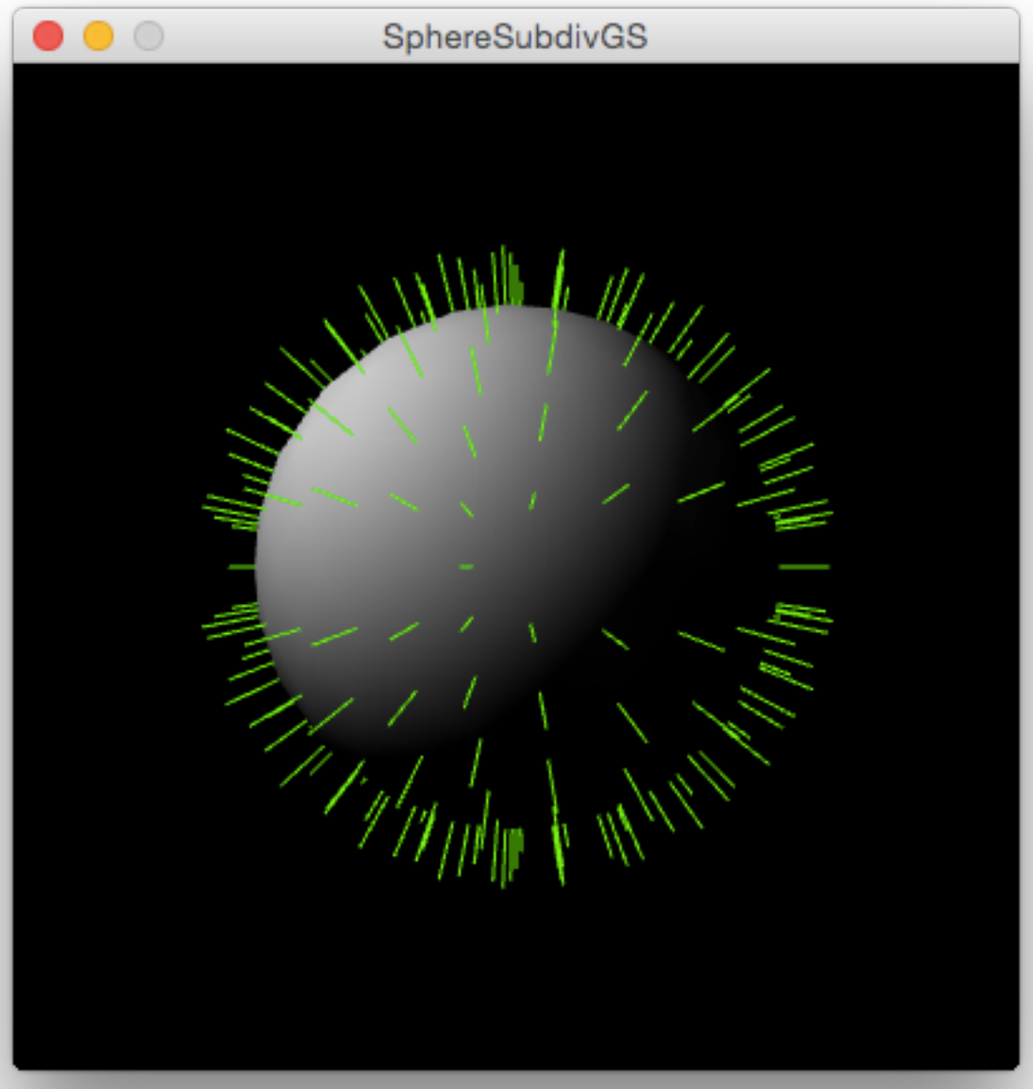
Just Cause 2 Square Enix

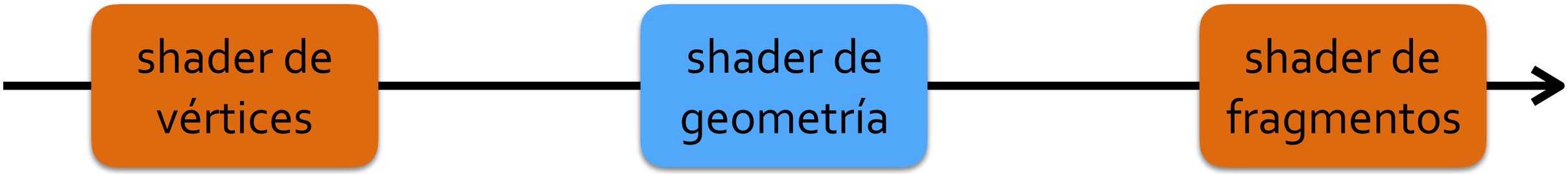


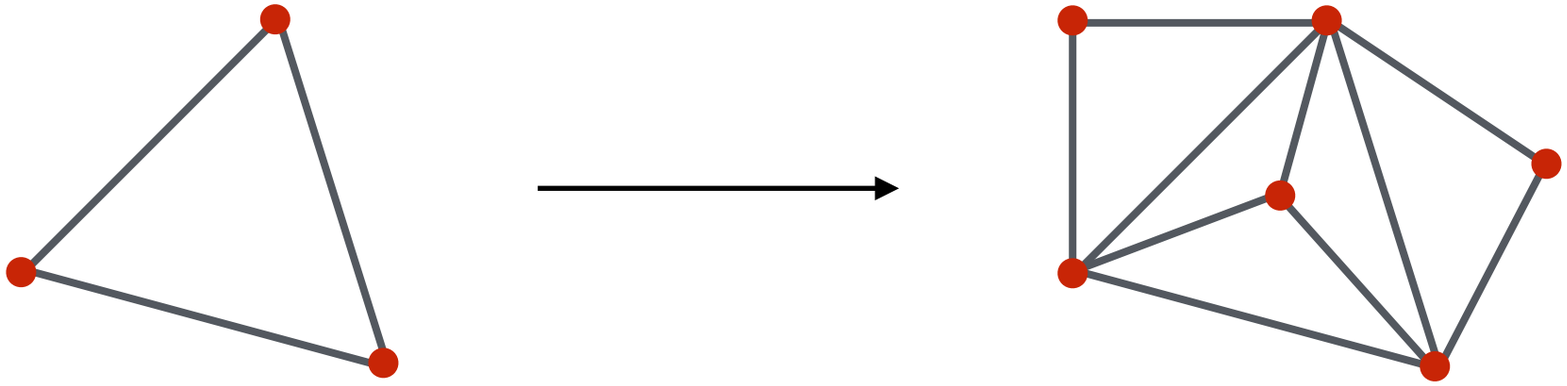
www.youtube.com/embed/hEoxaGkNcrg

Shaders de geometría (OpenGL 3)









geometría input (triangle)

geometría output (triangle_strip)

Uso de OpenGL “profiles” en Processing 3

Processing usa por defecto el profile 2, aun si los drivers de OpenGL soportan versiones más recientes

```
import com.jogamp.opengl.GL3;

void settings() {
  size(400, 400, P3D);
  PJOGL.profile = 3;
}

void setup() {
  octa = createOctahedron();

  subdiv = loadGeometryShader("vert.glsl", "geom.glsl", "frag.glsl");
  shader(subdiv);
}

void draw() {
  background(0);
```

Extendiendo la clase PShader

```
class GeometryShader extends PShader {
  int glGeometry;
  String geometrySource;

  GeometryShader(PApplet parent, String vertFilename,
                                 String geoFilename,
                                 String fragFilename) {
    super(parent, vertFilename, fragFilename);
    geometrySource = PApplet.join(parent.loadStrings(geoFilename), "\n");
  }

  void setup() {
    glGeometry = pgl.createShader(GL3.GL_GEOMETRY_SHADER);
    pgl.shaderSource(glGeometry, geometrySource);
    pgl.compileShader(glGeometry);

    pgl.getShaderiv(glGeometry, PGL.COMPILE_STATUS, intBuffer);
    boolean compiled = intBuffer.get(0) == 0 ? false : true;
    if (!compiled) {
      println("Cannot compile geometry shader:\n" + pgl.getShaderInfoLog(glGeometry));
      return;
    }

    pgl.attachShader(glProgram, glGeometry);
  }
}
```

GLSL version 1.50

vertice → geometría → fragmento

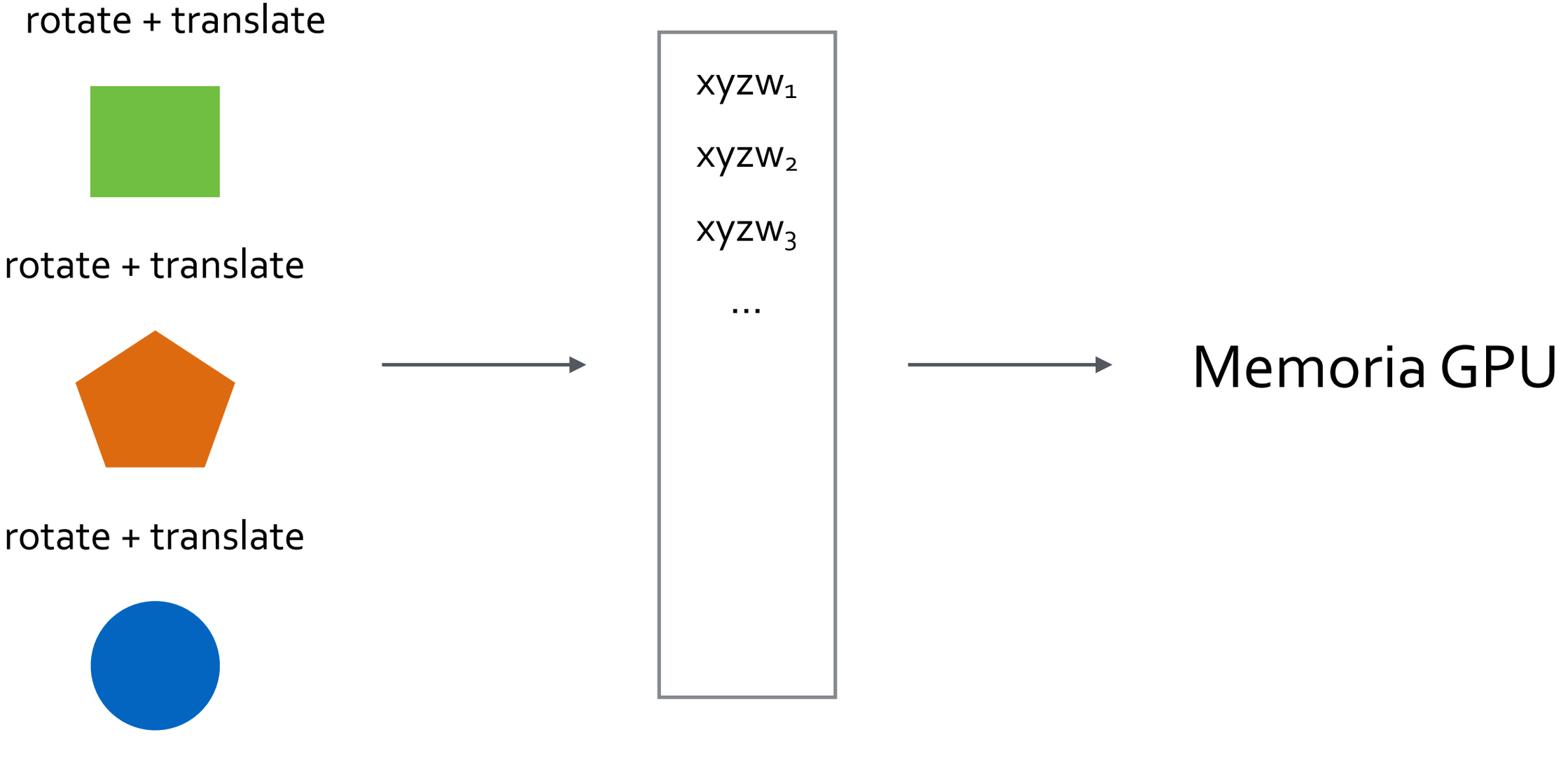
```
#version 150

in vec4 position;
in vec4 color;

out VertData {
  vec4 color;
} vertData;
```

```
#version 150

uniform mat4 modelview;
uniform mat4 transform;

in VertData {
  vec4 color;
} vertData[];

out FragData {
  vec4 color;
} fragData;
```

```
#version 150

in FragData {
  vec4 color;
} fragData;

out vec4 fragColor;
```

Mode inmediato vs retenido (PShape)

Las coordenadas de los vértices en modo inmediato ya están multiplicadas por la matriz modelview. ¿Porque?



Esto tiene como consecuencia que la matrix modelview
es la identidad cuando se dibujan shapes en modo inmediato,
y es la matrix que contiene las transformaciones de cámara de y de
modelo cuando se dibujan shapes en modo retenido (PShape)

```
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```

→ `modelview = identity`

```
s = createShape();
s.beginShape();
s.fill(0, 0, 255);
s.noStroke();
s.vertex(0, 0);
s.vertex(0, 50);
s.vertex(50, 50);
s.vertex(50, 0);
s.endShape(CLOSE);
```

→ `modelview = model * view`

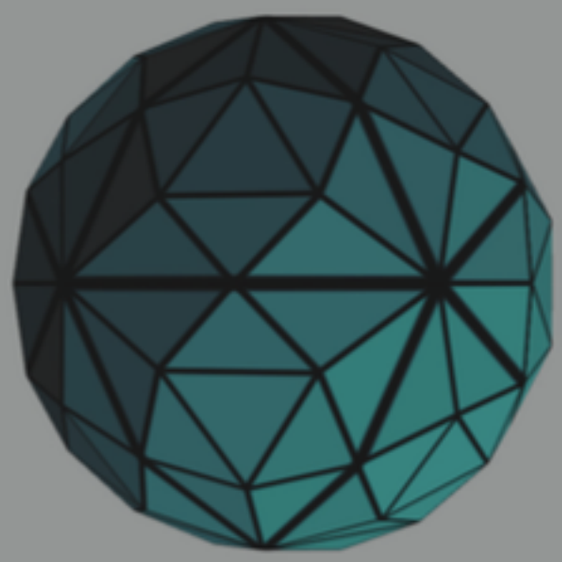
Limitaciones del shader de geometría

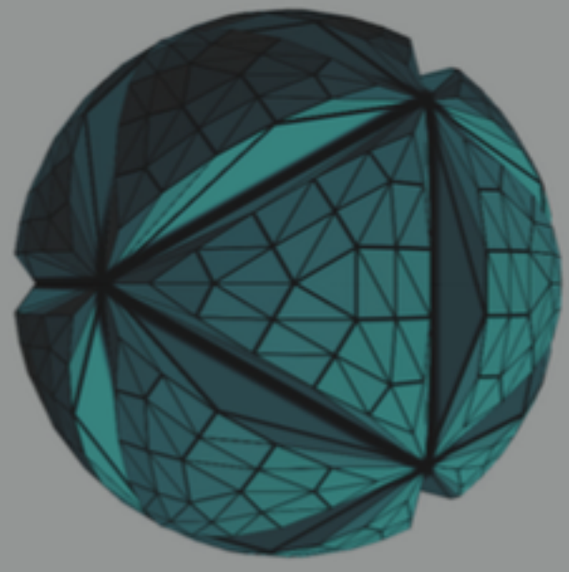
↓

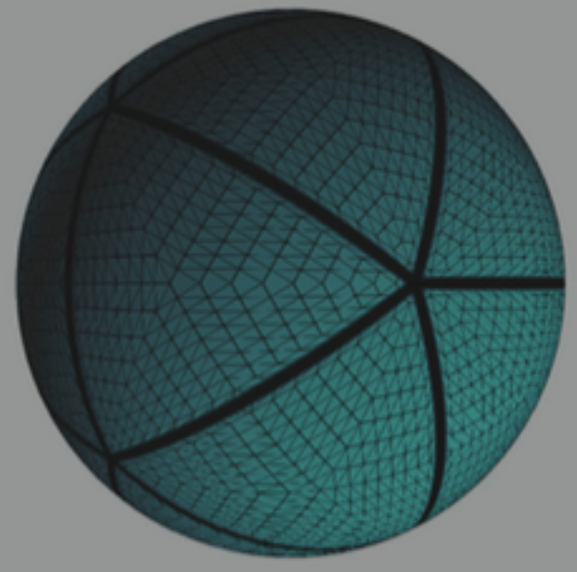
No paraleliza la emisión de nuevas primitivas

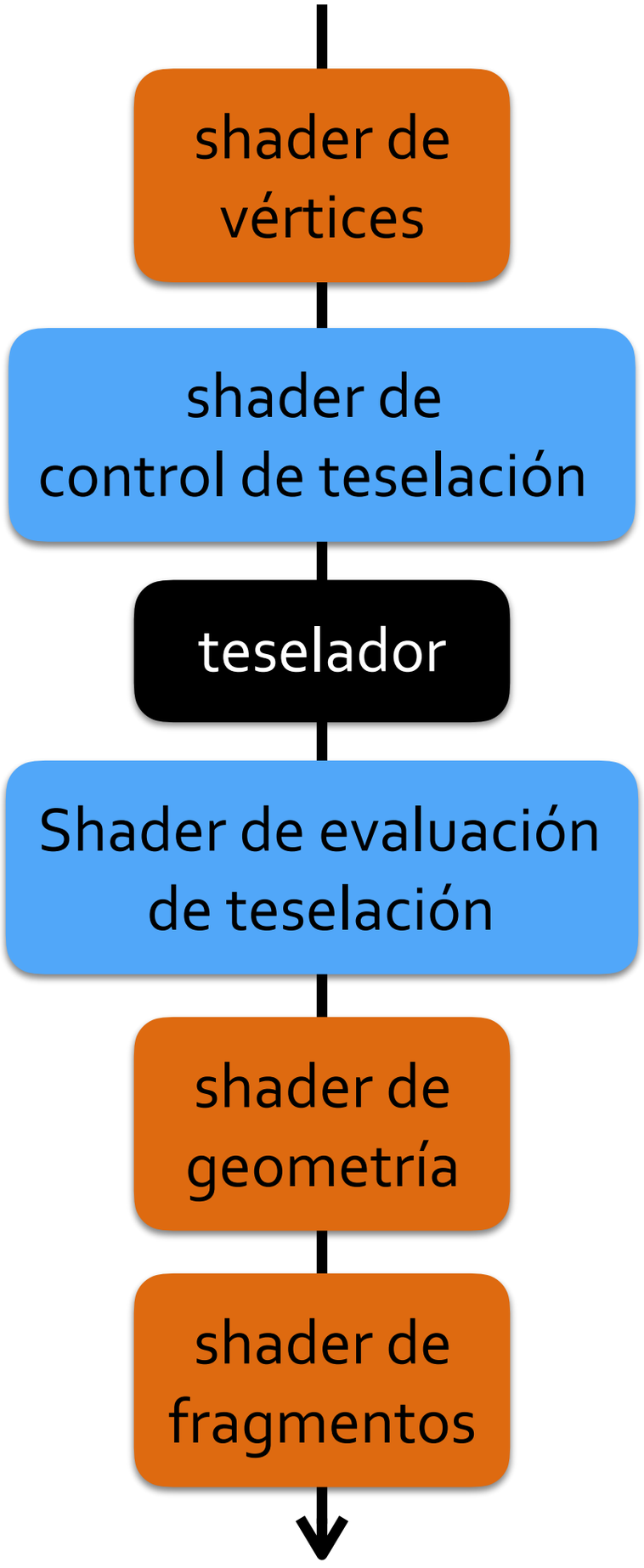
↓

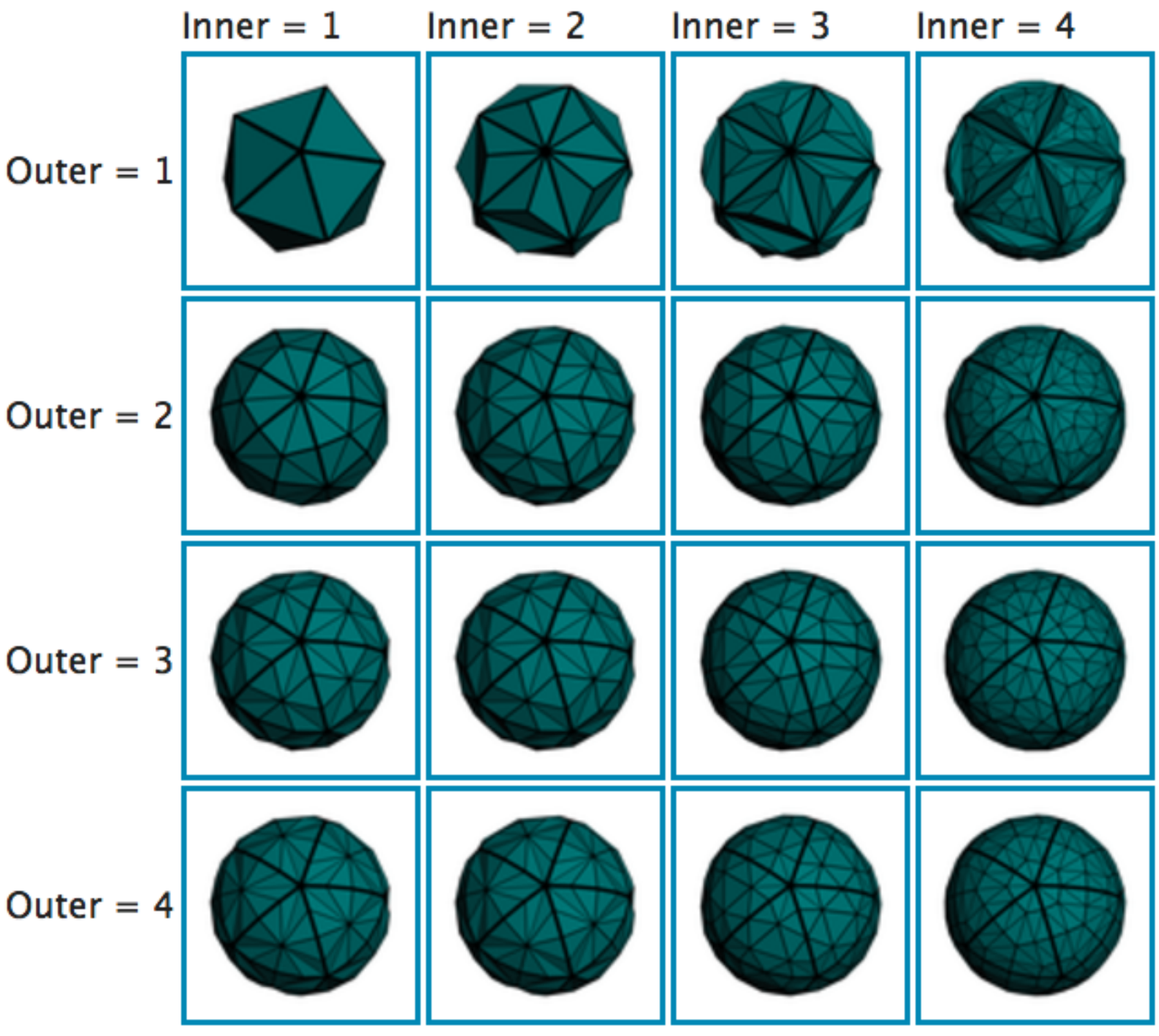
Shaders de teselación (OpenGL 4)








shader de vértices
shader de control de teselación
teselador
Shader de evaluación de teselación
shader de geometría
shader de fragmentos

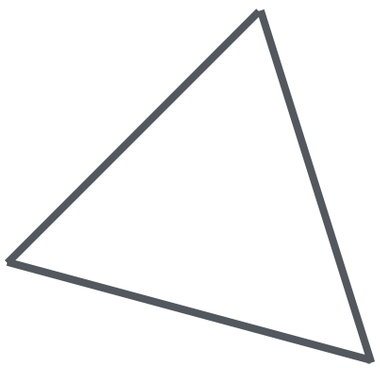


http://prideout.net/blog/?p=48

control de teselación

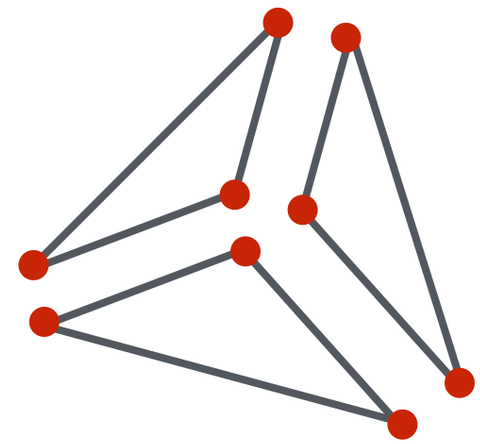
teselador

evaluación de teselación



"patch"

número de subdivisiones



primitivas generadas en paralelo

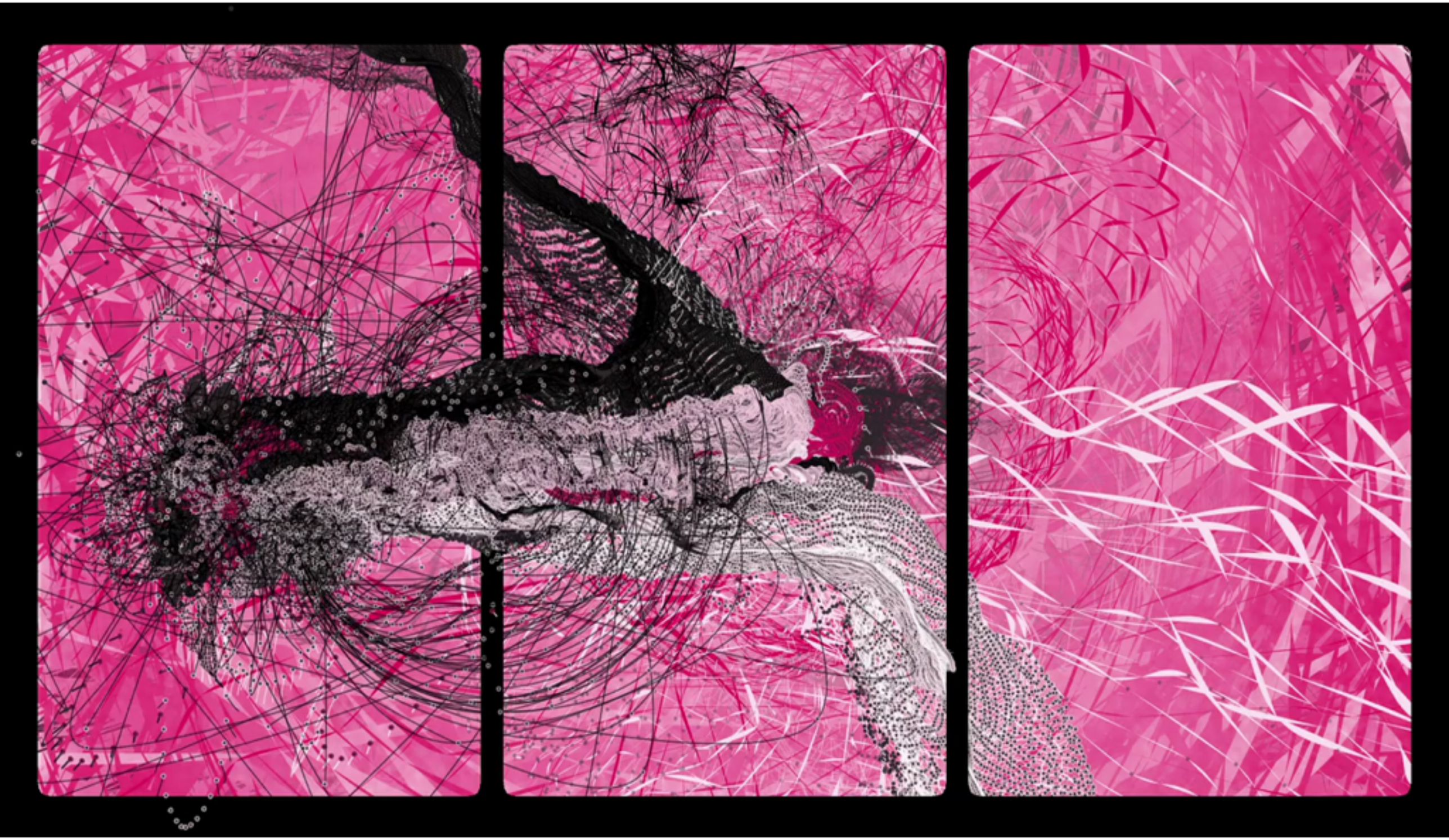
https://raw.githubusercontent.com/SaschaWillems/Vulkan/master/triangle/triangle.cpp

```
/*
* Vulkan Example - Basic indexed triangle rendering
*
* Note:
*	This is a "pedal to the metal" example to show off how to get Vulkan up an displaying something
*	Contrary to the other examples, this one won't make use of helper functions or initializers
*	Except in a few cases (swap chain setup e.g.)
*
* Copyright (C) 2016 by Sascha Willems - www.saschawillems.de
*
* This code is licensed under the MIT license (MIT) (http://opensource.org/licenses/MIT)
*/
```

¡1000 líneas de código!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <vector>
#include <exception>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <vulkan/vulkan.h>
#include "vulkanexamplebase.h"

#define VERTEX_BUFFER_BIND_ID 0
// Set to "true" to enable Vulkan's validation layers (see vulkandebug.cpp for details)
#define ENABLE_VALIDATION false
// Set to "true" to use staging buffers for uploading vertex and index data to device local memory
// See "prepareVertices" for details on what's staging and on why to use it
#define USE_STAGING true

class VulkanExample : public VulkanExampleBase
{
public:
	// Vertex buffer and attributes
	struct {
		VkDeviceMemory memory;
Handle to the device memory for this buffer
		VkBuffer buffer;
// Handle to the Vulkan buffer object that the memory is bound to
		VkPipelineVertexInputStateCreateInfo inputState;
```

Alba Corral: Performance AV en tiempo real



https://vimeo.com/albagcorral