Fakt: Die Computergrafik Pipeline, die auf praktisch jeder modernen GPU zur Bildsynthese eingesetzt wird, ist eine *Object Order* Technik.

Erklären Sie, was ein *Object Order Algorithmus* überhaupt ist, und wie sich dies konkret in der CG Pipeline bemerkbar macht.

Wie funktioniert im Gegensatz dazu ein Image Order Algorithmus, und welches entsprechende Prinzip zur Bildsynthese kennen Sie?

Kennen Sie Vor- und Nachteile der beiden Ansätze?

Sie kennen im Großen und Ganzen* zwei OpenGL Funktionen, um Primitive zu rendern:

- `glDrawArrays`
- `glDrawElements`

Erklären Sie, was das für einen Unterschied macht: In was für einer Struktur müssen die Daten der Objekte jeweils abgelegt sein? Erklären Sie das anhand eines einfachen geometrischen Objektes.

Warum ist es in der Praxis meistens von Vorteil, glDrawElements zu verwenden?

Erklären Sie, was man beim Rendern unter *interleaved* bzw. *non-interleaved* Vertex Data versteht.

*Es gibt eine Fülle an OpenGL Befehlen zum Rendern von Geometrie. Im Prinzip bauen aber alle auf diesen beiden Basis Modi auf, weils hier um die zugrunde liegende Datenstruktur geht.

Sie möchten ein 1024x1024 Bild invertieren. Aus Gründen der Performance entscheiden Sie, diesauf der GPU in einem Fragment Shader durchzuführen.

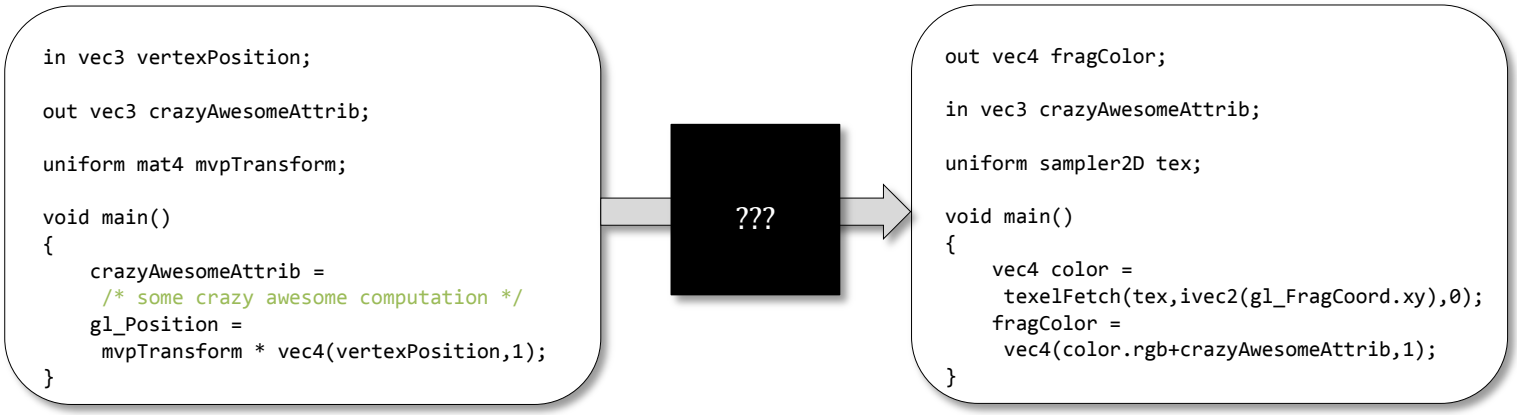
Ihr Fragment Shader wird demzufolge wohl so (bzw. so ähnlich) aussehen:

```
#version 330 core

uniform sampler2D image;
out vec4 fragColor;

void main()
{
	ivec2 pixel = ivec2( gl_FragCoord.xy );
	vec4 val = texelFetch( image, pixel, 0 );	// fetch pixel
	fragColor = vec4( 1.0 ) - val;			// invert
}
```

Erklären Sie, wie Sie erzwingen können, dass dieser Fragment Shader für jedes Pixel im Bild genau einmal aufgerufen wird: Was für Geometrie rendern Sie? Wie sieht der entsprechende Vertex Shader aus? Welche OpenGL Settings müssen Sie noch korrekt festlegen?



Zweierlei Daten (Vertex Attribute) werden im obigen Fall aus dem Vertex Shader zum Fragment Shader weitergeleitet: built-in *gl_Position* & das selbst-deklarierte *crazyAwesomeAttrib*. Allerdings liegen zwischen den beiden Shadern ja noch diverse andere, nicht programmierbare GPU Stages.

- ➔ Was macht die GPU zwischen Vertex & Fragment Shader mit *gl_Position*? Was kommt im Fragment Shader letztlich an?
- ➔ Und wie verhält es sich mit *crazyAwesomeAttrib*?

Wenn man eine Vertex Normale in den World Space transformiert – zum Beispiel weil man damit Phong Shading berechnen möchte - sieht das in GLSL üblicherweise so aus:

Vertex Shader:

```
out vec3 normal;
uniform mat4 modelTf, viewTf, projectionTf;

void main()
{
	mat3 normalTf =
	     inverse( transpose( mat3( modelTf ) ) );
	normal = normalize( normalTf * vertexNormal );

	…
}
```

Fragment Shader:

```
in vec3 normal;

void main()
{
	vec3 N = normalize( normal );

	…
}
```

Warum darf man hier die Model Matrix von 4x4 auf 3x3 casten? Man entfernt damit ja schließlich einen Teil der Transformation.

Welchen Zweck hat es, die inverse-transponierte der (auf 3x3 gecasteten) Model Matrix zu verwenden? Was könnte passieren, wenn man diesen Teil weglässt?

Warum ruft man zwei Mal *normalize* auf – sowohl im Vertex, als auch im Fragment Shader?

Im obigen Beispiel wird die Normale in den sog. *World Space* transformiert. Was wäre anders, die Normale in den Eye Space transformiert werden sollte?

An welcher Stage der CG Pipeline wird der Z-Test durchgeführt, und wie genau wird er durchgeführt? Was versteht man unter *Early-Z Optimization*?

Fakt: Der Z-Buffer ist nicht linear. Was genau bedeutet das, und was für Probleme ergeben sich daraus?

Welche Methoden fürs Hidden Surface Removal kennen Sie, abgesehen vom Z-Buffer? Erklären Sie!