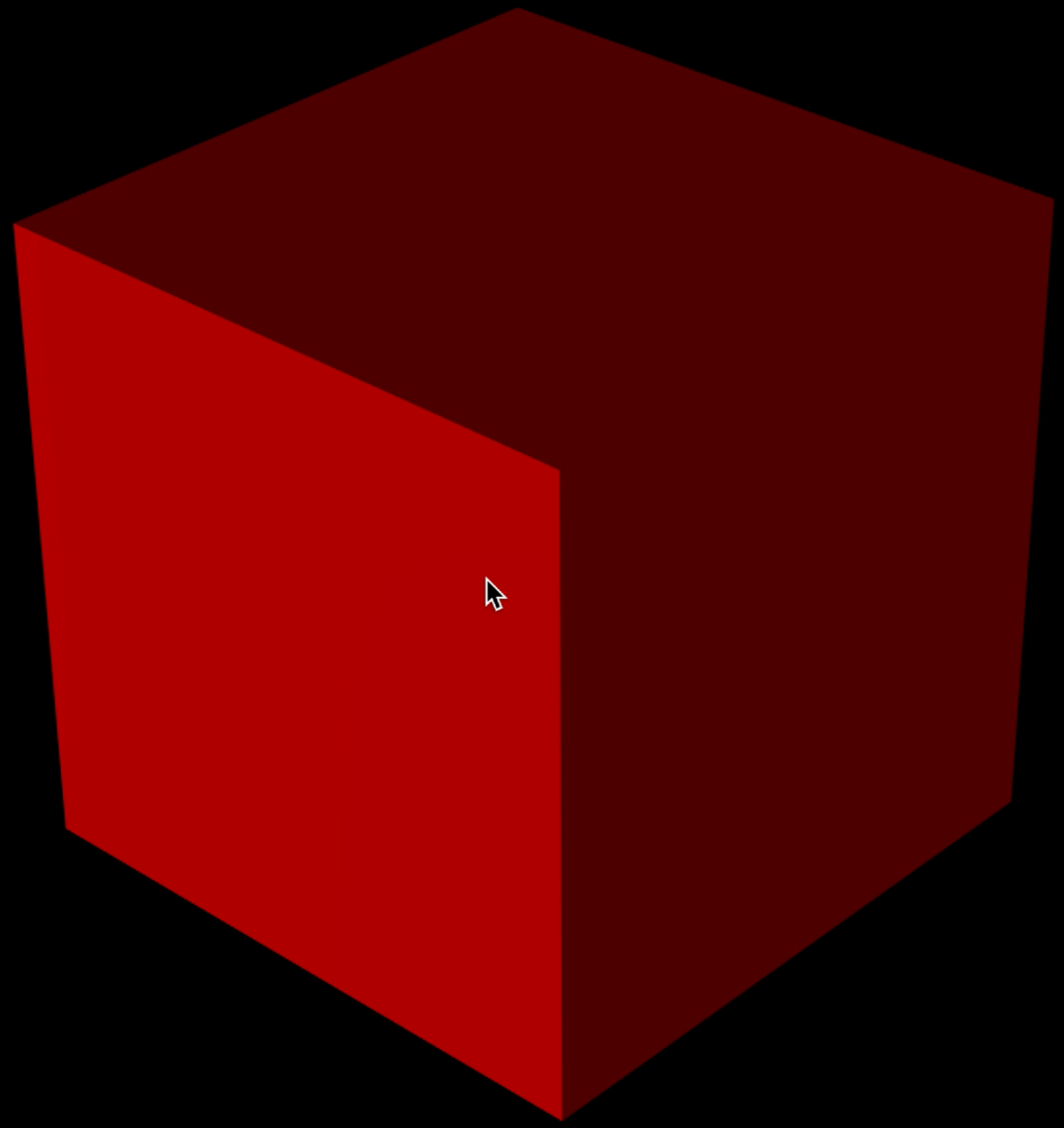
Рендерим 3D по-взрослому

Артем Кунец


.2
.4
.4
.4
.4
.4
.8
.9
.2
.2
.2
.2
.2
1.1




U.S. AIR FORCE

3D – это сложно!

3D в браузере

- Игры
- Статичный рендер
- Специальные эффекты
- 360 панорамы
- 3D редакторы
- 2D интерфейс
- 3D модели

3D движок

Пользовательский ввод

Геометрия

Свет

Частицы

Анимации

Материалы

Сцена

Морфинг

Текстуры

Шейдер

Скелетная анимация

Камера

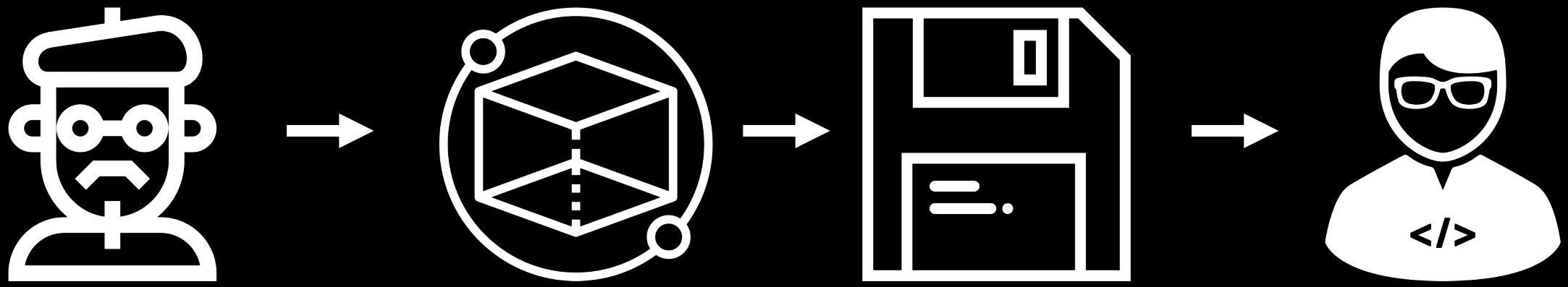
Пост обработка

Библиотека с математикой

Зачем писать свой

- Долго изучать, быстрее написать
- Нет подходящих решений, например смарт часы или ТВ
- Слишком универсален, накладывает ограничения
- Изучить технологию

С чего начать



Художник

Сцена

Экспорт

Разработчик

```
  # Список вершин, с координатами (x,y,z[,w]), w является не обязательным и по
умолчанию 1.0.
  v 0.123 0.234 0.345 1.0
  v ...
  ...
  # Текстурные координаты (u,v[,w]), w является не обязательным и по умолчанию 0.
  # Текстурная координата по y может быть указана как 1 - v, и при этом по x = u
  vt 0.500 -1.352 [0.234]
  vt ...
  ...
  # Нормали (x,y,z); нормали могут быть не .
.
  vn 0.707 0.000 0.707
  vn ...
  ...
  # Параметры вершин в пространстве (u [,v] [,w]); свободная форма геометрического
состояния (смотри ниже)
  vp 0.310000 3.210000 2.100000
  vp ...
  ...
  # Определения поверхности (сторон) (смотри ниже)
  f 1 2 3
  f 3/1 4/2 5/3
  f 6/4/1 3/5/3 7/6/5
  f 6//1 3//3 7//5
  f ...
  ...
  # Группа
  g Group1
  ...
  # Объект
  o Object1
```

Obj

```
<library_geometries>
    <geometry id="LOD3spShape-lib" name="LOD3spShape">
        <mesh>
            <source id="LOD3spShape-lib-positions" name="position">
                <technique_common>
                    <accessor count="2108" offset="0" source="#LOD3spShape-lib-positions-array" stride="3">
                        <param name="X" type="float"/>
                        <param name="Y" type="float"/>
                        <param name="Z" type="float"/>
                    </accessor>
                </technique_common>
            </source>
            <source id="LOD3spShape-lib-normals" name="normal">
                <technique_common>
                    <accessor count="2290" offset="0" source="#LOD3spShape-lib-normals-array" stride="3">
                        <param name="X" type="float"/>
                        <param name="Y" type="float"/>
                        <param name="Z" type="float"/>
                    </accessor>
                </technique_common>
            </source>
            <source id="LOD3spShape-lib-map1" name="map1">
                <technique_common>
                    <accessor count="2277" offset="0" source="#LOD3spShape-lib-map1-array" stride="2">
                        <param name="S" type="float"/>
                        <param name="T" type="float"/>
                    </accessor>
                </technique_common>
            </source>
            <vertices id="LOD3spShape-lib-vertices">
                <input semantic="POSITION" source="#LOD3spShape-lib-positions"/>
            </vertices>
            <polylist count="2144" material="blinn3SG">
                <input offset="0" semantic="VERTEX" source="#LOD3spShape-lib-vertices"/>
                <input offset="1" semantic="NORMAL" source="#LOD3spShape-lib-normals"/>
                <input offset="2" semantic="TEXCOORD" source="#LOD3spShape-lib-map1" set="0"/>
            </polylist>
        </mesh>
    </geometry>
</library_geometries>
<library_visual_scenes>
```

Collada

GLTF (GL Transmission Format) 2.0

Протокол передачи 3D объектов.

- Это спецификация от Khronos Group
- JSON
- Большие данные в двоичном формате
- Поддерживает все, что требуется современному движку
- Изначально ориентирован на веб и WebGL
- Расширяемый через дополнения

GLTF

Спецификация:

https://github.com/KhronosGroup/glTF/blob/master/specification/2.0/README.md

Набор тестовых моделей:

https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0

Список движков:

https://github.com/cx20/gltf-test

glTF - what the ?

An overview of the basics of the GL Transmission Format

glTF was designed and specified by the Khronos Group, for the efficient transfer of 3D content over networks.

The core of glTF is a **JSON** file that describes the structure and composition of a scene containing 3D models. The top-level elements of this file are:

scenes, nodes
Basic structure of the scene

cameras
View configurations for the scene

meshes
Geometry of 3D objects

buffers, bufferViews, accessors
Data references and data layout descriptions

materials
Definitions of how objects should be rendered

textures, images, samplers
Surface appearance of objects

skins
Information for vertex skinning

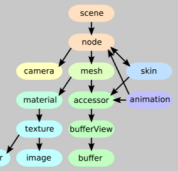
animations
Changes of properties over time

These elements are contained in arrays. References between the objects are established by using their indices to look up the objects in the arrays.

It is also possible to store the whole asset in a single binary glTF file. In this case, the JSON data is stored as a string, followed by the binary data of buffers or images.

Concepts

The conceptual relationships between the top-level elements of a glTF asset are shown here:



Binary data references

The images and buffers of a glTF asset may refer to external files that contain the data that are required for rendering the 3D content:

```
"buffers": [
  {
    "uri": "buffer01.bin",
    "byteLength": 102040,
  }
],
"images": [
  {
    "uri": "image01.png"
  }
],
```

The **buffers** refer to binary files (.BIN) that contain geometry- or animation data.

The **images** refer to image files (PNG, JPG...) that contain texture data for the models.

The data is referred to via URIs, but can also be included directly in the JSON using data URIs. The data URI defines the MIME type, and contains the data as a base64 encoded string:

Buffer data:

```
"data:application/gltf-buffer;base64,AAABAAIAAgA..."
```

Image data (PNG):

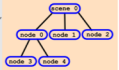
```
"..."
```

scenes, nodes

The glTF JSON may contain **scenes** (with an optional default **scene**). Each scene can contain an array of indices of nodes.

```
"scene": 0,
"scenes": [
  {
    "nodes": [ 0, 1, 2 ]
  }
],
"nodes": [
  {
    "children": [ 3, 4 ],
  },
  { ... },
  { ... },
  { ... },
  ...
],
```

Each of the **nodes** can contain an array of indices of its **children**. This allows modeling a simple scene hierarchy:




```
"nodes": [
  {
    "matrix": [
      1,0,0,0,
      0,1,0,0,
      0,0,1,0,
      5,6,7,1
    ],
    ...
  },
  {
    "translation": [ 0,0,0 ],
    "rotation": [ 0,0,0,1 ],
    "scale": [ 1,1,1 ]
    ...
  },
]
```

A node may contain a local transform. This can be given as a column-major **matrix** array, or with separate **translation**, **rotation** and **scale** properties, where the rotation is given as a quaternion. The local transform matrix is then computed as

M = T * R * S

where **T**, **R** and **S** are the matrices that are created from the translation, rotation and scale. The global transform of a node is given by the product of all local transforms on the path from the root to the respective node.

```
"nodes": [
  {
    "mesh": 4,
    ...
  },
  {
    "camera": 2,
    ...
  }
]
```

Each node may refer to a **mesh** or a **camera**, using indices that point into the meshes and cameras arrays. These elements are then attached to these nodes. During rendering, instances of these elements are created and transformed with the global transform of the node.

The translation, rotation and scale properties of a node may also be the target of an animation: The animation can describe how one of the property changes over time. The attached objects will move accordingly, allowing to model moving objects or camera flights.

Nodes are also used in vertex skinning: A node hierarchy can define the skeleton of an animated character. The node then refers to a mesh and to a skin. The skin contains further information about how the mesh is deformed based on the current skeleton pose.

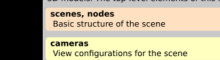
meshes

The **meshes** may contain multiple mesh **primitives**. These refer to the geometry data that is required for rendering the mesh.

```
"meshes": [
  {
    "primitives": [
      {
        "mode": 4,
        "indices": 0,
        "attributes": {
          "POSITION": 1,
          "NORMAL": 2
        },
        "material": 2
      }
    ]
  },
]
```

Each mesh primitive has a rendering **mode**, which is a constant indicating whether it should be rendered as **POINTS**, **LINES**, or **TRIANGLES**. The primitive also refers to **indices** and the **attributes** of the vertices, using the indices of the accessors for this data. The **material** that should be used for rendering is also given, by the index of the material.

Each attribute is defined by mapping the attribute name to the index of the accessor that contains the attribute data. This data will be used as the vertex attributes when rendering the mesh. The attributes may, for example, define the **POSITION** and the **NORMAL** of the vertices:



A mesh may define multiple morph targets. Such a morph target describes a deformation of the original mesh.

```
"primitives": [
  {
    ...
    "targets": [
      {
        "POSITION": 11,
        "NORMAL": 13
      },
      {
        "POSITION": 21,
        "NORMAL": 23
      }
    ]
  }
],
"weights": [ 0, 0.5]
```

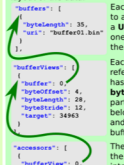
To define a mesh with morph targets, each mesh primitive can contain an array of **targets**. These are dictionaries that map names of attributes to the indices of accessors that contain the displacements of the geometry for the target.

The mesh may also contain an array of **weights** that define the contribution of each morph target to the final, rendered state of the mesh.

Combining multiple morph targets with different weights allows, for example, modeling different facial expressions of a character: The weights can be modified with an animation, to interpolate between different states of the geometry.

buffers, bufferViews, accessors

The **buffers** contain the data that is used for the geometry of 3D models, animations, and skinning. The **bufferViews** add structural information to this data. The **accessors** define the exact type and layout of the data.



```
"buffers": [
  {
    "byteLength": 35,
    "uri": "buffer01.bin"
  }
],
"bufferViews": [
  {
    "buffer": 0,
    "byteOffset": 4,
    "byteLength": 28,
    "byteStride": 12,
    "target": 34963
  }
],
"accessors": [
  {
    "bufferView": 0,
    "byteOffset": 4,
    "type": "VEC2",
    "componentType": 5126,
    "count": 2,
    "min": [0.1, 0.2]
    "max" : [0.9, 0.8]
  }
]
```

Each of the **buffers** refers to a binary data file, using a **URI**. It is the source of one block of raw data with the given **byteLength**.

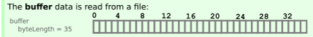
Each of the **bufferViews** refers to one buffer. It has a **byteOffset** and a **byteLength**, defining the part of the buffer that belongs to the bufferView, and an optional OpenGL buffer **target**.

The **accessors** define how the data of a bufferView is interpreted. They may define an additional **byteOffset** referring to the start of the bufferView, and contain information about the type and layout of the bufferView data:

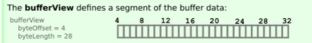
The data may, for example, be defined as 2D vectors of floating point values when the **type** is **"VEC2"** and the **componentType** is GL_FLOAT (5126). The range of all values is stored in the **min** and **max** property.

The data of multiple accessors may be interleaved inside a bufferView. In this case, the bufferView will have a **byteStride** property that says how many bytes are between the start of one element of an accessor, and the start of the next.

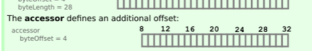
The **buffer** data is read from a file:



The **bufferView** defines a segment of the buffer data:

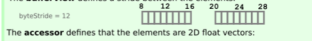


The **accessor** defines an additional offset:



The **bufferView** defines a stride between the elements:

The **accessor** defines that the elements are 2D float vectors:



This data may, for example, be used by a mesh primitive, to access 2D texture coordinates: The data of the **bufferView** may be bound as an OpenGL buffer, using **glBindBuffer**. Then, the properties of the **accessor** may be used to define this buffer as vertex attribute data, by passing them to **glVertexAttribPointer** when the bufferView buffer is bound.

Sparse accessors

When only few elements of an **accessor** differ from a default value (which is often the case for morph targets), then the data can be given in a very compact form using a **sparse** data description:

```
"accessors": [
  {
    "type": "SCALAR",
    "componentType": 5126,
    "count": 10,
    "sparse": {
      "count": 4,
      "values": {
        "bufferView": 2,
      },
      "indices": {
        "bufferView": 1,
        "componentType": 5123
      }
    }
  }
]
```

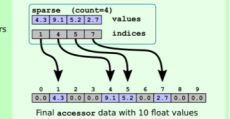
The accessor defines the type of the data (here, scalar float values), and the total element **count**.

The **sparse** data block contains the **count** of sparse data elements.

The **values** refer to the bufferView that contains the sparse data values.

The target **indices** for the sparse data values are defined with a reference to a bufferView and the **componentType**.

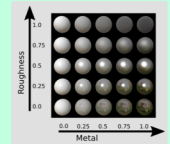
The **values** are written into the final accessor data, at the positions that are given by the **indices**:



materials

Each mesh primitive may refer to one of the **materials** that are contained in a glTF asset. The materials describe how an object should be rendered, based on physical material properties. This allows to apply **Physically Based Rendering (PBR)** techniques, to make sure that the appearance of the rendered object is consistent among all renderers.

The default material model is the **Metallic-Roughness-Model**. Values between 0.0 and 1.0 are used to describe how much the material characteristics resemble that of a metal, and how rough the surface of the object is. These properties may either be given as individual values that apply to the whole object, or be read from textures.




```
"materials": [
  {
    "pbrMetallicRoughness": {
      "baseColorTexture": {
        "index": 1,
        "texCoord": 1
      },
      "baseColorFactor":
        [ 1.0, 0.75, 0.35, 1.0 ],
      "metallicRoughnessTexture": {
        "index": 5,
        "texCoord": 1
      },
      "metallicFactor": 1.0,
      "roughnessFactor": 0.0,
    }
    "normalTexture": {
      "scale": 0.8,
      "index": 2,
      "texCoord": 1
    },
    "occlusionTexture": {
      "strength": 0.9,
      "index": 4,
      "texCoord": 1
    },
    "emissiveTexture": {
      "index": 3,
      "texCoord": 1
    },
    "emissiveFactor":
      [0.4, 0.8, 0.6]
  }
],
```

The properties that define a material in the Metallic-Roughness-Model are summarized in the **pbrMetallicRoughness** object:

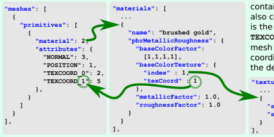
The **baseColorTexture** is the main texture that will be applied to the object. The **baseColorFactor** contains scaling factors for the red, green, blue and alpha component of the color. If no texture is used, these values will define the color of the whole object.

The **metallicRoughnessTexture** contains the metalness value in the "blue" color channel, and the roughness value in the "green" color channel. The **metallicFactor** and **roughnessFactor** are multiplied with these values. If no texture is given, then these factors define the reflection characteristics for the whole object.

In addition to the properties that are defined via the Metallic-Roughness-Model, the material may contain other properties that affect the object appearance:

- The **normalTexture** refers to a texture that contains tangent-space normal information, and a **scale** factor that will be applied to these normals.
- The **occlusionTexture** refers to a texture that defines areas of the surface that are occluded from light, and thus rendered darker. This information is contained in the "red" channel of the texture. The occlusion **strength** is a scaling factor to be applied to these values.
- The **emissiveTexture** refers to a texture that may be used to illuminate parts of the object surface. It defines the color of the light that is emitted from the surface. The **emissiveFactor** contains scaling factors for the red, green and blue components of this texture.

Material properties in textures



```
"meshes": [
  {
    "primitives": [
      {
        "material": 2,
        "attributes": {
          "NORMAL": 3,
          "POSITION": 1,
          "TEXCOORD_0": 2,
          "TEXCOORD_1": 5
        },
      }
    ]
  },
],
"materials": [
  ...
  {
    "name": "brushed gold",
    "pbrMetallicRoughness": {
      "baseColorFactor":
        [1,1,1,1],
      "baseColorTexture": {
        "index": 1,
        "texCoord": 1
      },
      "metallicFactor": 1.0,
      "roughnessFactor": 1.0
    }
  }
],
```

The texture references in a material always contain the **index** of the texture. They may also contain the **texCoord** set index. This is the number that determines the **TEXCOORD_<n>** attribute of the rendered mesh primitive that contains the texture coordinates for this texture, with 0 being the default.

```
"textures": [
  ...
  {
    "source": 4,
    "sampler": 2
  }
],
```

cameras

Each of the nodes may refer to one of the **cameras** that are defined in the glTF asset.

```
"cameras": [
  {
    "type": "perspective",
    "perspective": {
      "aspectRatio": 1.5,
      "yfov": 0.65,
      "zfar": 100,
      "znear": 0.01
    }
  },
  {
    "type": "orthographic",
    "orthographic": {
      "xmag": 1.0,
      "ymag": 1.0,
      "zfar": 100,
      "znear": 0.01
    }
  }
]
```

There are two types of cameras: **perspective** and **orthographic** ones, and they define the projection matrix.

The value for the far clipping plane distance of a perspective camera, **zfar**, is optional. When it is omitted, the camera uses a special projection matrix for infinite projections.

When one of the nodes refers to a camera, then an instance of this camera is created. The camera matrix of this instance is given by the global transform matrix of the node.

textures, images, samplers

The **textures** contain information about textures that may be applied to rendered objects: Textures are referred to by materials to define the basic color of the objects, as well as physical properties that affect the object appearance.

```
"textures": [
  {
    "source": 4,
    "sampler": 2
  }
],
"images": [
  {
    "uri": "file01.png"
  }
  {
    "bufferView": 3,
    "mimeType":
      "image/jpeg"
  }
],
"samplers": [
  {
    "magFilter": 9729,
    "minFilter": 9987,
    "wrapS": 10497,
    "wrapT": 10497
  }
],
```

The texture consists of a reference to the **source** of the texture, which is one of the **images** of the asset, and a reference to a **sampler**.

The **images** define the image data used for the texture. This data can be given via a **URI** that is the location of an image file, or by a reference to a **bufferView** and a **MIME type** that defines the type of the image data that is stored in the buffer view.

The **samplers** describe the wrapping and scaling of textures. (The constant values correspond to OpenGL constants that can directly be passed to **glTexParameter**).

skins

A glTF asset may contain the information that is necessary to perform vertex skinning. With vertex skinning, it is possible to let the vertices of a mesh be influenced by the bones of a skeleton, based on its current pose.

```
"nodes": [
  {
    "name": "Skinned mesh node",
    "mesh": 0
    "skin": 0
  },
  {
    "name": "Torso",
    "children": [ 2, 3, 4, 5, 6 ],
    "rotation": [...],
    "scale": [...],
    "translation": [...]
  },
  {
    "name": "LegL",
    "children": [ 7 ],
  },
  {
    "name": "FootL",
    ...
  },
  ...
],
```

A node that refers to a **mesh** may also refer to a **skin.**

```
"skins": [
  {
    "inverseBindMatrices": 12,
    "joints": [ 1, 2, 3 ... ]
  },
],
```

The **skins** contain an array of **joints**, which are the indices of nodes that define the skeleton hierarchy, and the **inverseBindMatrices,** which is a reference to an accessor that contains one matrix for each joint.

The skeleton hierarchy is modeled with nodes, just like the scene structure: Each joint node may have a local transform and an array of children, and the "bones" of the skeleton are given implicitly, as the connections between the joints.

The mesh primitives of a skinned mesh contain the **POSITION** attribute that refers to the accessor for the vertex positions, and two special attributes that are required for skinning: A **JOINTS_0** and a **WEIGHTS_0** attribute, each referring to an accessor.

The **JOINTS_0** attribute data contains the indices of the joints that should affect the vertex.

The **WEIGHTS_0** attribute data defines the weights indicating how strongly the joint should influence the vertex.

From this information, the **skinning matrix** can be computed.

This is explained in detail in **"Computing the skinning matrix"**.

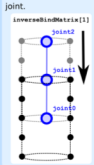
```
"meshes": [
  {
    "primitives": [
      {
        "attributes": {
          "POSITION": 0,
          "JOINTS_0": 1,
          "WEIGHTS_0": 2
        ...
      },
    ]
  }
],
```

Computing the skinning matrix

The skinning matrix describes how the vertices of a mesh are transformed based on the current pose of a skeleton. The skinning matrix is a weighted combination of joint matrices.

Computing the joint matrices

The skin refers to the **inverseBindMatrices**. This is an accessor which contains one inverse bind matrix for each joint. Each of these matrices transforms the mesh into the local space of the joint.



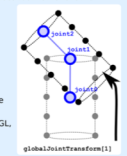
From these matrices, a **jointMatrix** may be computed for each joint:

```
jointMatrix[j] =
    inverse(globalTransform) *
    globalJointTransform[j] *
    inverseBindMatrix[j];
```

Any global transform of the node that contains the mesh and the skin is cancelled out by pre-multiplying the joint matrix with the inverse of this transform.

For implementations based on OpenGL or WebGL, the **jointMatrix** array will be passed to the vertex shader as a uniform.

For each node whose index appears in the **joints** of the skin, a global transform matrix can be computed. It transforms the mesh from the local space of the joint, based on the current global transform of the joint, and is called **globalJointTransform**.



Combining the joint matrices to create the skinning matrix

The primitives of a skinned mesh contain the **POSITION**, **JOINT** and **WEIGHT** attributes, referring to accessors. These accessors contain one element for each vertex:

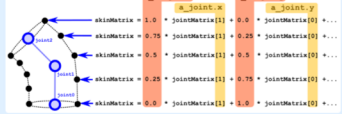
The data of these accessors is passed as attributes to the vertex shader, together with the **jointMatrix** array.

In the vertex shader, the **skinMatrix** is computed. It is a linear combination of the joint matrices whose indices are contained in the **JOINTS_0** attribute, weighted with the **WEIGHTS_0** values:

Vertex Shader

```
uniform mat4 u_jointMatrix[12];
attribute vec3 a_position;
attribute vec4 a_joint;
attribute vec4 a_weight;
...
void main(void) {
  ...
  mat4 skinMatrix =
    a_weight.x * u_jointMatrix[int(a_joint.x)] +
    a_weight.y * u_jointMatrix[int(a_joint.y)] +
    a_weight.z * u_jointMatrix[int(a_joint.z)] +
    a_weight.w * u_jointMatrix[int(a_joint.w)];
  gl_Position =
    modelViewProjection * skinMatrix * position;
}
```

The **skinMatrix** transforms the vertices based on the skeleton pose, before they are transformed with the model-view-perspective matrix.



Wiki page about skinning in COLLADA: https://www.khronos.org/collada/wiki/Skinning
Section 4-7 in the COLLADA specification: https://www.khronos.org/files/collada_spec_1_5.pdf
(The vertex skinning in COLLADA is similar to that in glTF)

animations

A glTF asset can contain **animations**. An animation can be applied to the properties of a node that define the local transform of the node, or to the weights for the morph targets.

```
"animations": [
  {
    "channels": [
      {
        "target": {
          "node": 1,
          "path": "translation"
        },
        "sampler": 0
      }
    ],
    "samplers": [
      {
        "input": 4,
        "interpolation": "LINEAR",
        "output": 5
      }
    ]
  }
]
```

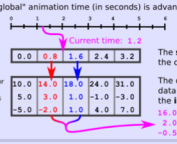
Each animation consists of two elements: An array of **channels** and an array of **samplers**.

Each channel defines the **target** of the animation. This target usually refers to a **node**, using the index of this node, and to a **path**, which is the name of the animated property. The path may be **"translation"**, **"rotation"** or **"scale"**, affecting the local transform of the node, or **"weights"**, in order to animate the weights of the morph targets of the meshes that are referred to by the node. The channel also refers to a **sampler**, which summarizes the actual animation data.

A sampler refers to the **input** and **output** data, using the indices of accessors that provide the data. The input refers to an accessor with scalar floating-point values, which are the times of the key frames of the animation. The output refers to an accessor that contains the values for the animated property at the respective key frames. The sampler also defines an **interpolation** mode for the animation, which may be **"LINEAR"**, **"STEP"**, **"CATMULLROMSPLINE"**, or **"CUBICSPLINE"**.

Animation samplers

During the animation, a "global" animation time (in seconds) is advanced.



The **sampler** looks up the key frames for the current time, in the **input** data.

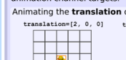
The corresponding values of the **output** data are read, and interpolated based on the **interpolation** mode of the sampler.

The interpolated value is forwarded to the animation channel target.

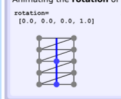
Animation channel targets

The interpolated value that is provided by an animation sampler may be applied to different animation channel targets.

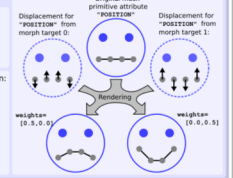
Animating the **translation** of a node:



Animating the **rotation** of a skeleton node of a skin:



Animating the **weights** for the morph targets that are defined for the primitives of a mesh that is attached to a node:



Binary glTF files

In the standard glTF format, there are two options for including external binary resources like buffer data and textures: They may be referenced via URIs, or embedded in the JSON part of the glTF using data URIs. When they are referenced via URIs, then each external resource implies a new download request. When they are embedded as data URIs, the base 64 encoding of the binary data will increase the file size considerably.

To overcome these drawbacks, there is the option to combine the glTF JSON and the binary data into a single **binary glTF** file. This is a little-endian file, with the extension **".glb"**. It contains a **header**, which gives basic information about the version and structure of the data, and one or more **chunks** that contain the actual data. The first chunk always contains the JSON data. The remaining chunks contain the binary data.

The **magic** entry has the value 0x46546C67, which is the ASCII string **"glTF"**. This is used to identify the data as a binary glTF

The **chunkLength** is the length of the **chunkData**, in bytes

The **version** defines the file format version. The version described here is version 2

The **chunkType** value defines what type of data is contained in the **chunkData**. It may be 0x4E4F534A, which is the ASCII string **"JSON"**, for JSON data, or 0x004E4942, which is the ASCII string **"BIN"**, for binary data.

The **length** is the total length of the file, in bytes

The **chunkData** contains the actual data of the chunk. This may be the ASCII representation of the JSON data, or binary buffer data.

Extensions

The glTF format allows extensions to add new functionality, or to simplify the definition of commonly used properties.

When an extension is used in a glTF asset, it has to be listed in the top-level **extensionsUsed** property. The **extensionsRequired** property lists the extensions that are strictly required to properly load the asset.

```
"extensionsUsed" : [
  "KHR_lights_common",
  "CUSTOM_EXTENSION"
]
"extensionsRequired" : [
  "KHR_lights_common"
]
```

```
"textures" : [
  ...
  "extensions" : {
    "KHR_lights_common" : {
      "lightSource" : true,
    },
    "CUSTOM_EXTENSION" : {
      "customProperty" :
        "customValue"
    }
  }
]
```

Extensions allow adding arbitrary objects in the **extensions** property of other objects.

The name of such an object is the same as the name of the extension, and it may contain further, extension-specific properties.

Existing extensions

The following extensions are developed and maintained on the Khronos GitHub repository:

- **Specular-Glossiness Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_materials_pbrSpecularGlossiness
  This extension is an alternative to the default Metallic-Roughness material model: It allows to define the material properties based on specular and glossiness values.
- **Common Materials**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_materials_cmnBlinnPhong
  This extension allows the easy definition of non-physically based materials. This is based on the Blinn-Phong model, which is often used in CAD applications. The material can be defined using a diffuse, specular, and emissive color, and a shininess value.
- **Common Lights**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_lights
  This extension allows adding different types of lights to the scene hierarchy. This refers to point lights, spot lights and directional lights. The lights can be attached to the nodes of the scene hierarchy.
- **WebGL Rendering Techniques**
  https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_technique_webgl
  With this extension, it is possible to define GLSL shaders that should be used for rendering the glTF asset in OpenGL or WebGL.

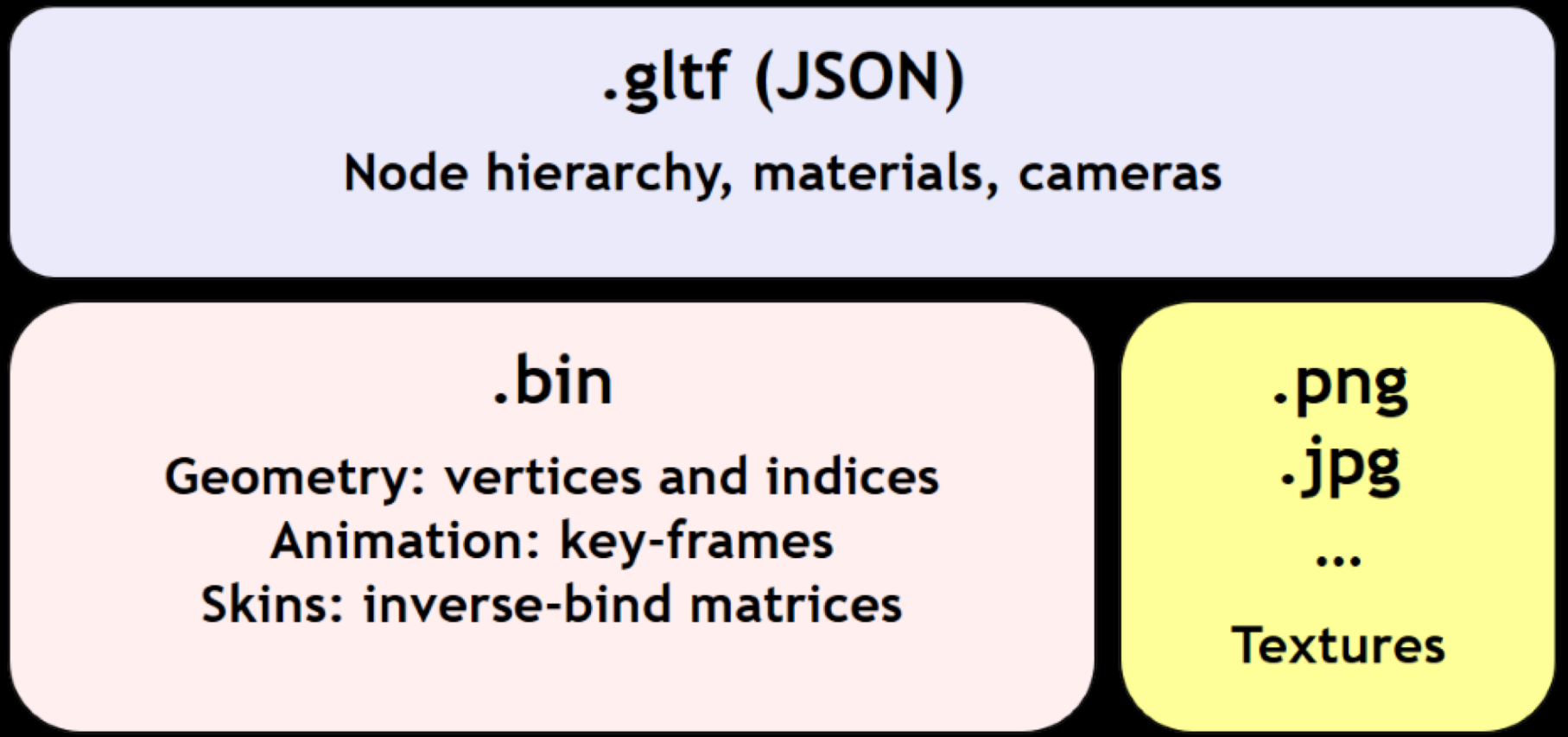
Further resources

The Khronos glTF landing page:
https://www.khronos.org/gltf

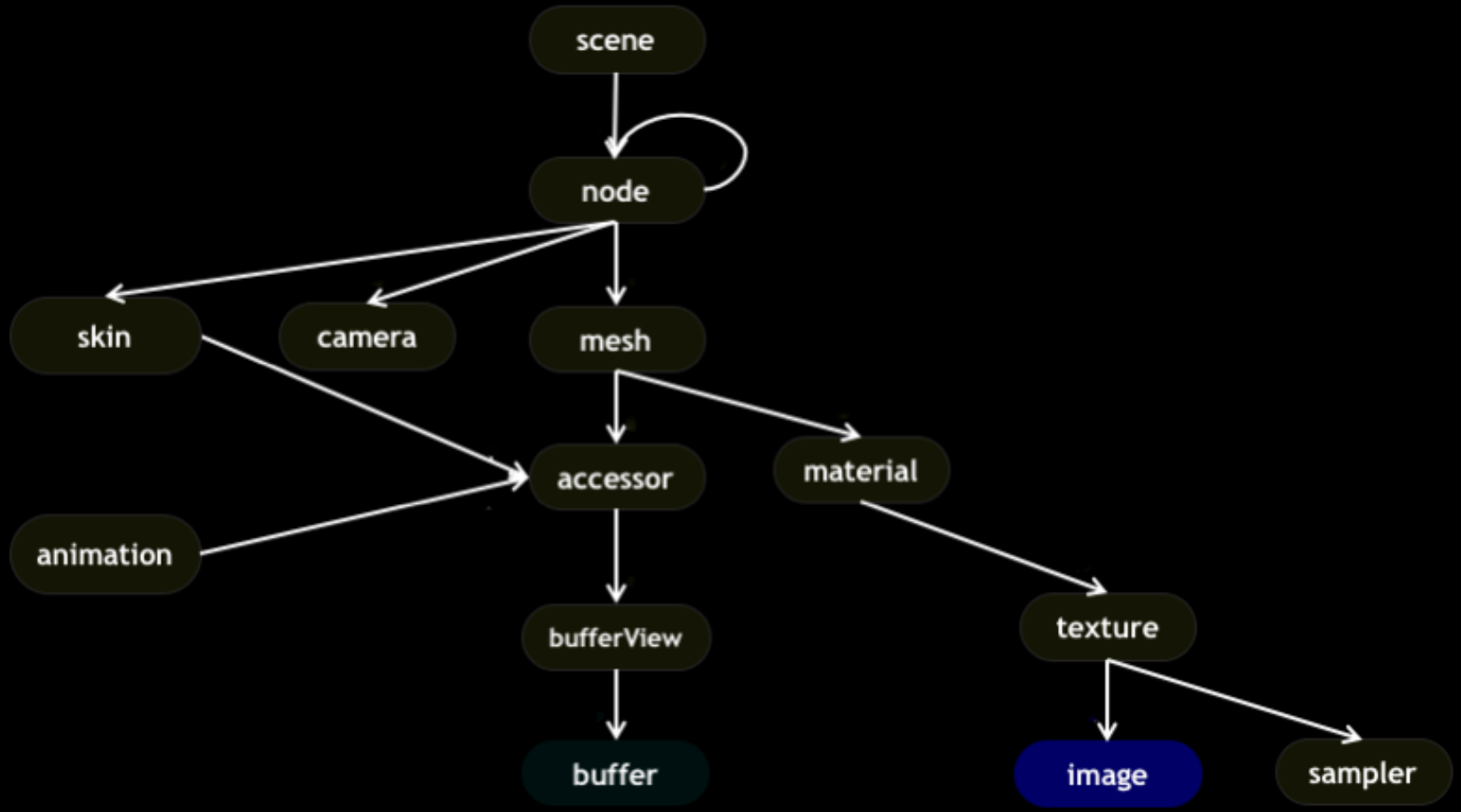
The Khronos glTF GitHub repository:
https://github.com/KhronosGroup/glTF

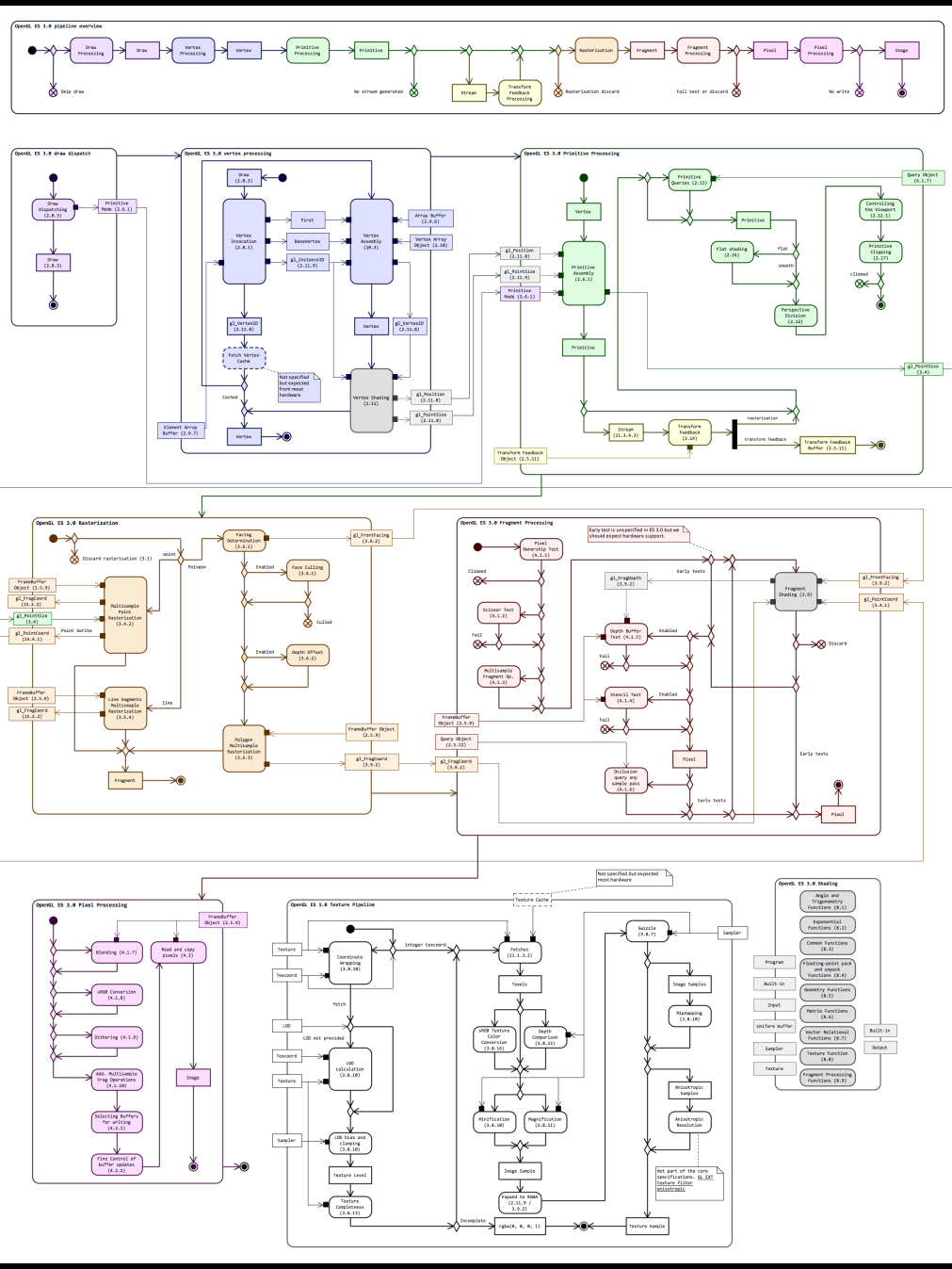
Version 2.0a
glTF version 2.0
This overview is non-normative!
Feedback:
glTF@marco-hutter.de

GLTF сцена


.gltf (JSON)
Node hierarchy, materials, cameras
.bin
Geometry: vertices and indices
Animation: key-frames
Skins: inverse-bind matrices
.png
.jpg
...
Textures

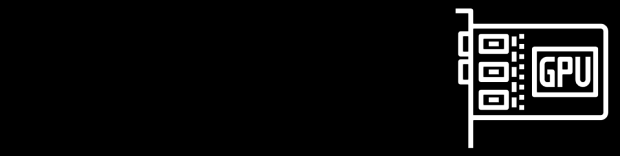
Структура


scene
node
skin
camera
mesh
accessor
material
animation
bufferView
texture
buffer
image
sampler

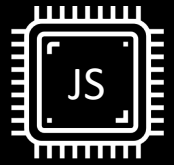

OpenGL ES 3.0 pipeline overview
Draw Processing
Draw
Vertex Processing
Vertex
Primitive Processing
Primitive
Rasterization
Fragment
Fragment Processing
Pixel
Pixel Processing
Image
Skip Draw
No stream generated
Stream
Transform Feedback Processing
Rasterization discard
Fail test or discard
No write
OpenGL ES 3.0 draw dispatch
Draw Dispatching (2.8.3)
Primitive Mode (2.8.1)
Draw (2.8.3)
OpenGL ES 3.0 vertex processing
Draw (2.8.3)
Vertex Invocation (2.8.1)
First
BaseVertex
gl_InstanceID (2.11.9)
Vertex Assembly (10.3)
Array Buffer (2.9.6)
Vertex Array Object (2.10)
gl_VertexID (2.11.8)
Fetch Vertex Cache
Not specified but expected from most hardware
Cached
Element Array Buffer (2.9.7)
Vertex
Vertex Shading (2.11)
gl_Position (2.11.8)
gl_PointSize (2.11.8)
OpenGL ES 3.0 Primitive Processing
Vertex
gl_Position (2.11.8)
gl_PointSize (2.11.9)
Primitive Mode (2.8.1)
Primitive Assembly (2.8.1)
Primitive
Primitive Queries (2.15)
Query Object (6.1.7)
Flat shading (2.16)
flat
smooth
Perspective Division (2.12)
Controlling the Viewport (2.12.1)
Primitive Clipping (2.13)
clipped
gl_PointSize (3.4)
Stream (11.3.4.3)
Transform Feedback (2.24)
rasterization
transform feedback
Transform Feedback Buffer (2.5.11)
Transform Feedback Object (2.5.11)
OpenGL ES 3.0 Rasterization
Discard rasterization (3.1)
point
Polygon
Facing Determination (3.6.1)
gl_FrontFacing (3.9.2)
Enabled
Face Culling (3.6.1)
Culled
Depth Offset (3.6.2)
FrameBuffer Object (2.5.9)
gl_FragCoord (3.9.2)
Multisample Point Rasterization (3.4.2)
gl_PointCoord (14.4.1)
Point Sorite
Line Segments Multisample Rasterization (3.5.4)
line
Polygon Multisample Rasterization (3.6.3)
Fragment
OpenGL ES 3.0 Fragment Processing
Early test is unspecified in ES 3.0 but we should expect hardware support.
Pixel Ownership Test (4.1.1)
Closed
Scissor Test (4.1.2)
Fail
Multisample Fragment Op. (4.1.3)
gl_FragDepth (3.9.2)
Early tests
Fragment Shading (3.9)
Depth Buffer Test (4.1.5)
Stencil Test (4.1.4)
Occlusion query any sample pass (4.1.6)
Pixel
Discard
gl_FrontFacing (3.9.2)
gl_PointCoord (9.4.1)
OpenGL ES 3.0 Pixel Processing
FrameBuffer Object (2.5.9)
Blending (4.1.7)
Read and copy pixels (4.3)
sRGB Conversion (4.1.8)
Dithering (4.1.9)
Add. Multisample Frag Operations (9.1.10)
Selecting Buffers for Writing (4.2.1)
Fine Control of Buffer updates (4.2.2)
Image
OpenGL ES 3.0 Texture Pipeline
Not specified but expected most hardware
Texture Cache
Texture
Texcoord
Coordinate Wrapping (3.8.10)
integer texcoord
Fetch
LOD
LOD not provided
LOD calculation (3.8.10)
Sampler
LOD bias and clamping (3.8.10)
Texture Level
Texture Completeness (3.8.13)
Incomplete
Fetches (11.1.3.2)
Texels
sRGB Texture Color Conversion (3.8.16)
Depth Comparison (3.8.15)
Minification (3.8.10)
Magnification (3.8.11)
Image Sample
Expand to RGBA (2.15.9 / 3.9.2)
rgba(0, 0, 0, 1)
Swizzle (3.8.7)
Image Samples
Mipmapping (3.8.10)
Anisotropic Samples
Anisotropic Resolution
Not part of the core specifications: GL_EXT_texture_filter_anisotropic
Texture Sample
OpenGL ES 3.0 Shading
Angle and Trigonometry Functions (8.1)
Exponential Functions (8.2)
Common Functions (8.3)
Floating-point pack and unpack Functions (8.4)
Geometry Functions (8.5)
Matrix Functions (8.6)
Vector Relational Functions (8.7)
Texture Function (8.8)
Fragment Processing Functions (8.9)
Program
Built-in
Input
Uniform buffer
Sampler
Texture
Built-in
Output

Что где

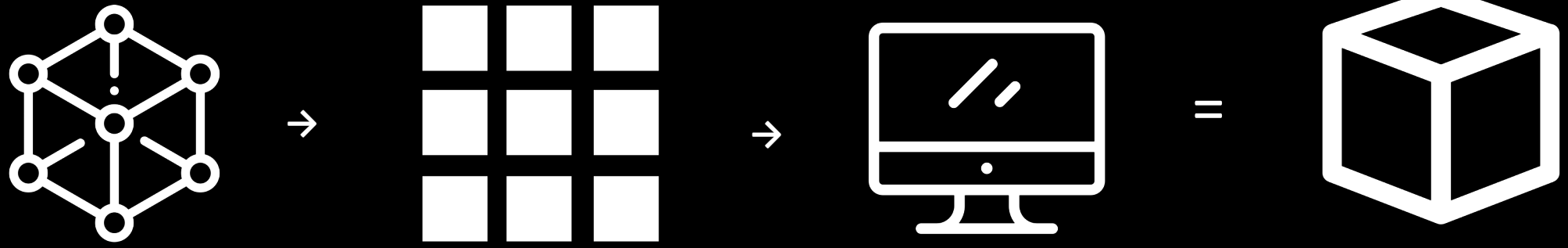
- Буферы
- Шейдеры
- Текстуры
- Состояние
- GLSL



- Создание и подготовка данных для GPU
- Управление состоянием GPU
- Вызов отрисовки
- JS



WebGL конвейер



Вершины

Пространство отсечения

Экранное пространство

Сборка примитива

Растеризация

Обработка пикселей

Тест пикселя

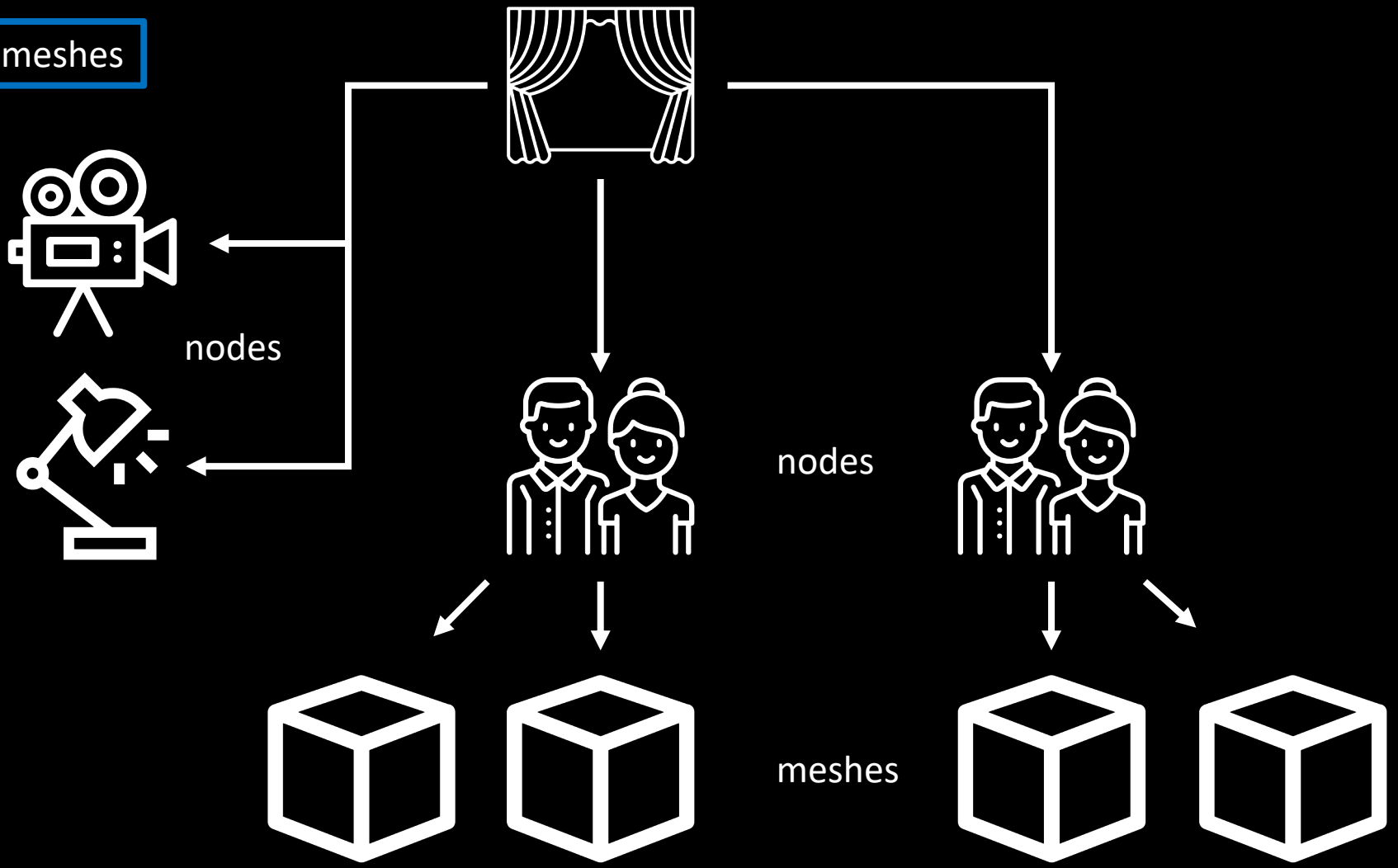
Фреймбуффер

Основные 3D объекты

- Сцена – структура всех 3D объектов
- Нода – объект сцены, имеет потомков
- Меш – объект с геометрией и материалом
- Камера – область сцены видимая на экране
- Источник света – объект используемый в модели освещения

Структура

gltf.nodes и gltf.meshes



Нода

```
interface Node {
    children: Array<Node | Mesh>;
    model: Matrix4;
    modelWorld: Matrix4;
    visible: boolean;
    parent: Node;
}
```

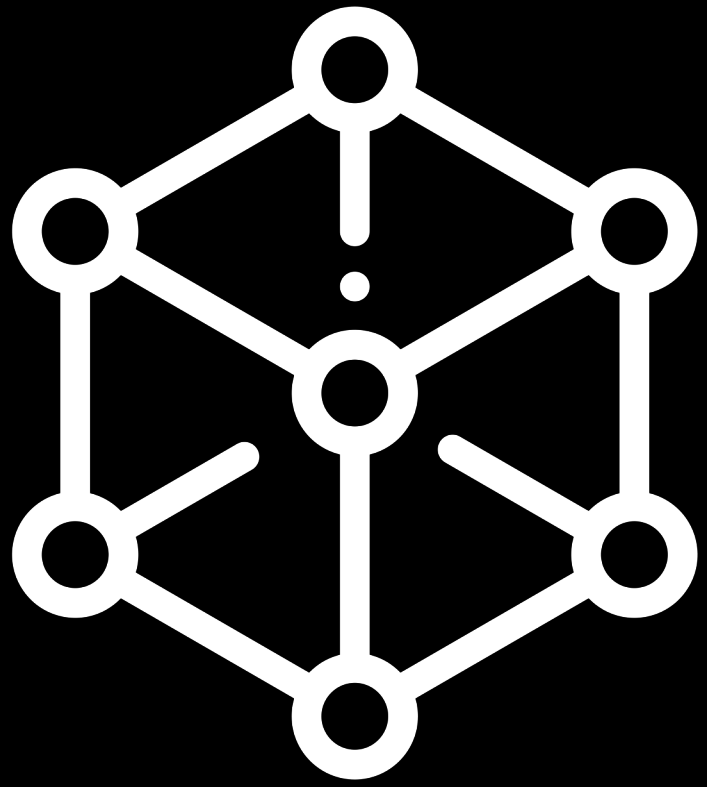
Меш

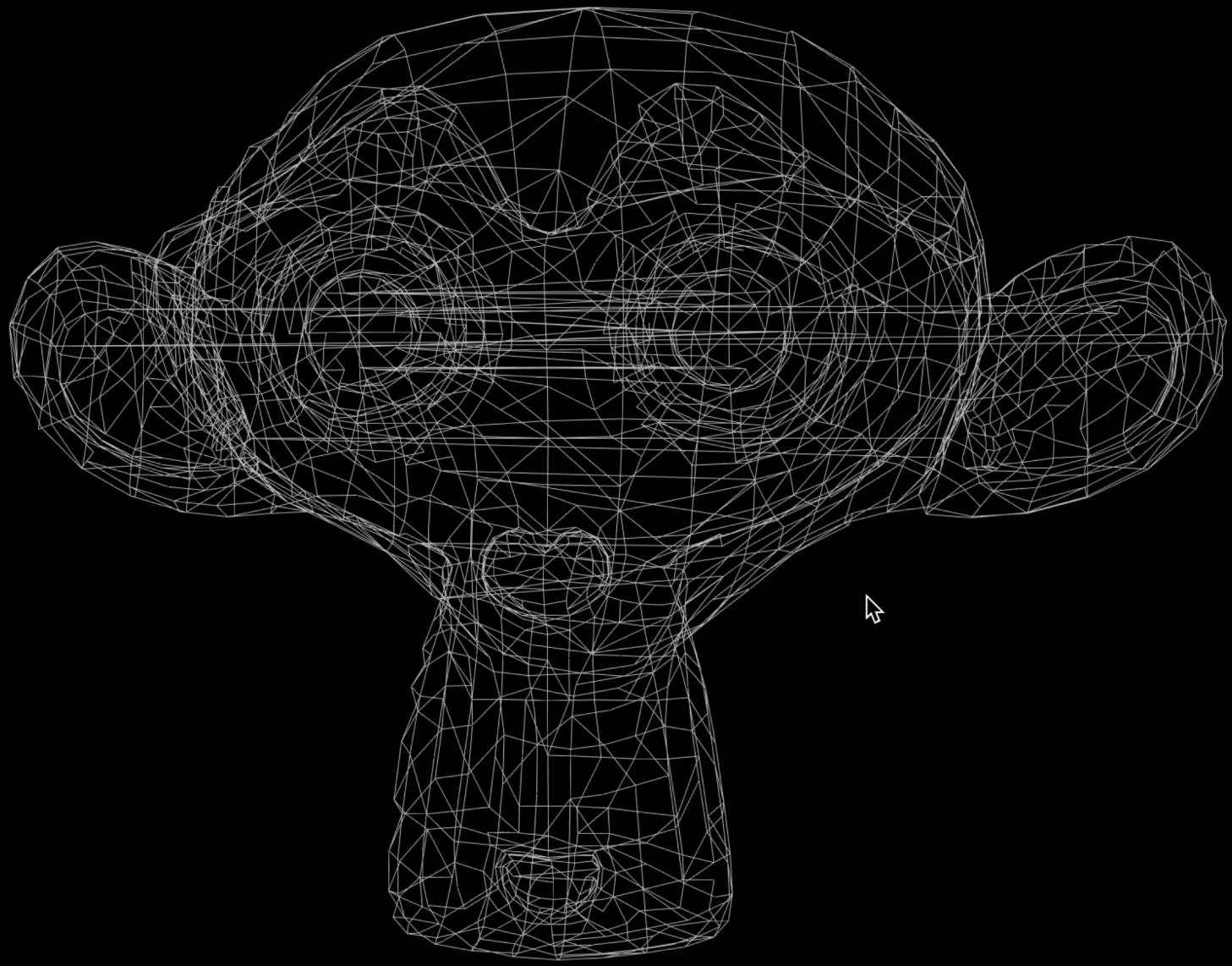
```
interface Mesh extends Node {
    geometry: Geometry;
    material: Material;
    program: WebGLProgram;
}
```

Геометрия

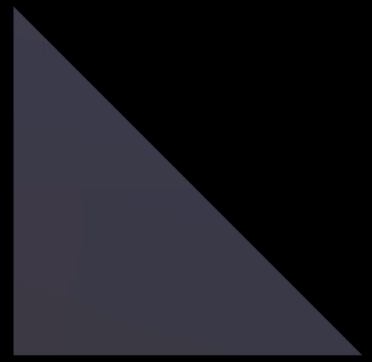
gltf.meshes.primitives -> gltf.accessors -> gltf.bufferViews -> gltf.buffers

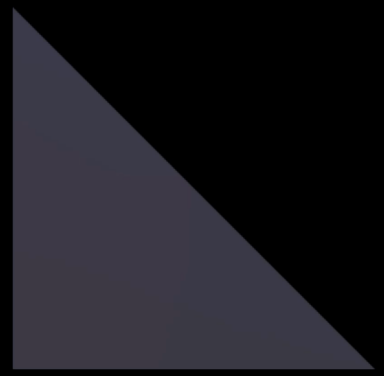
```
interface Geometry {
        indices: byte | short | int;
        vertices: Vec3;
        normals: Vec3;
        uv: Vec2;
}
```



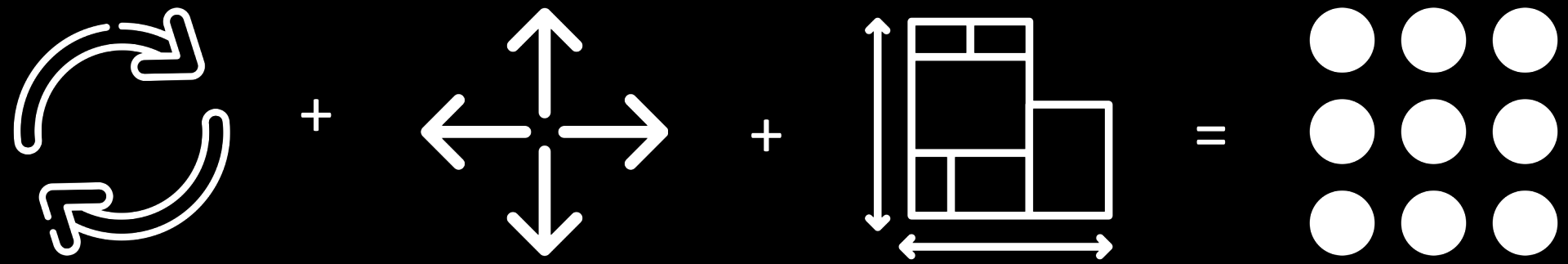


Трансформации



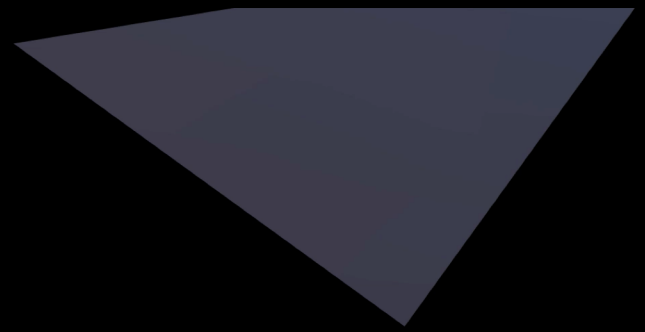


Трансформации



Вращение Перемещение Масштабирование Модельная матрица

Матрица трансформаций



Матрицы

```
const matrix = new Float32Array([
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
]);
```

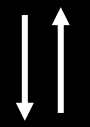
```
// вращение
matrix.rotate(Vector4);
// масштабирование
matrix.scale(Vector3);
// перемещение
matrix.translate(Vector3);

const x = matrix[12];
const y = matrix[13];
const z = matrix[14];
const translation = new Vector3(x, y, z);
```

Про матрицы

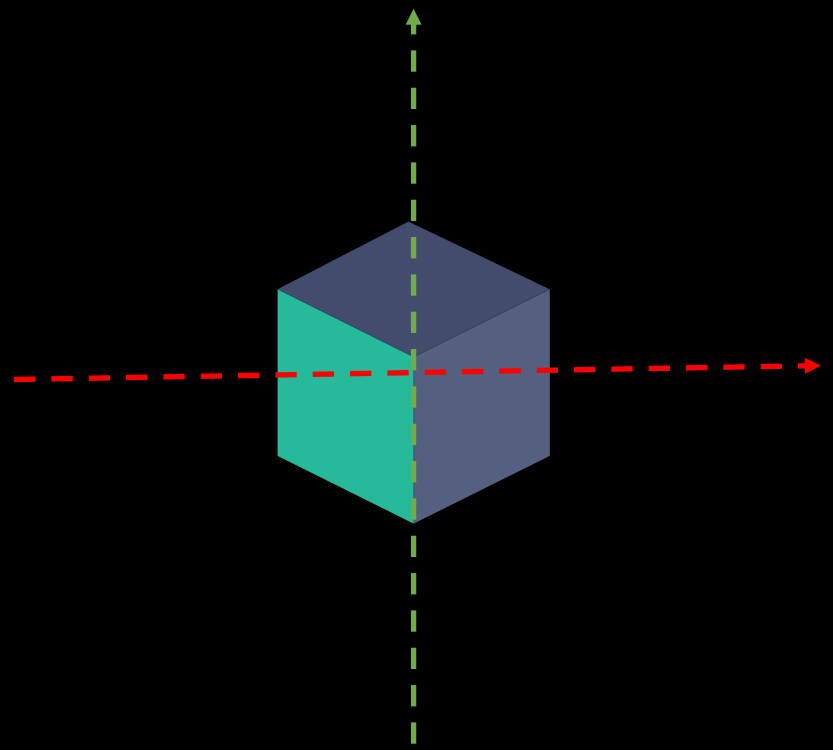
Матрица1 * Матрица2 !== Матрица2 * Матрица1

Матриа1 * Матрица2

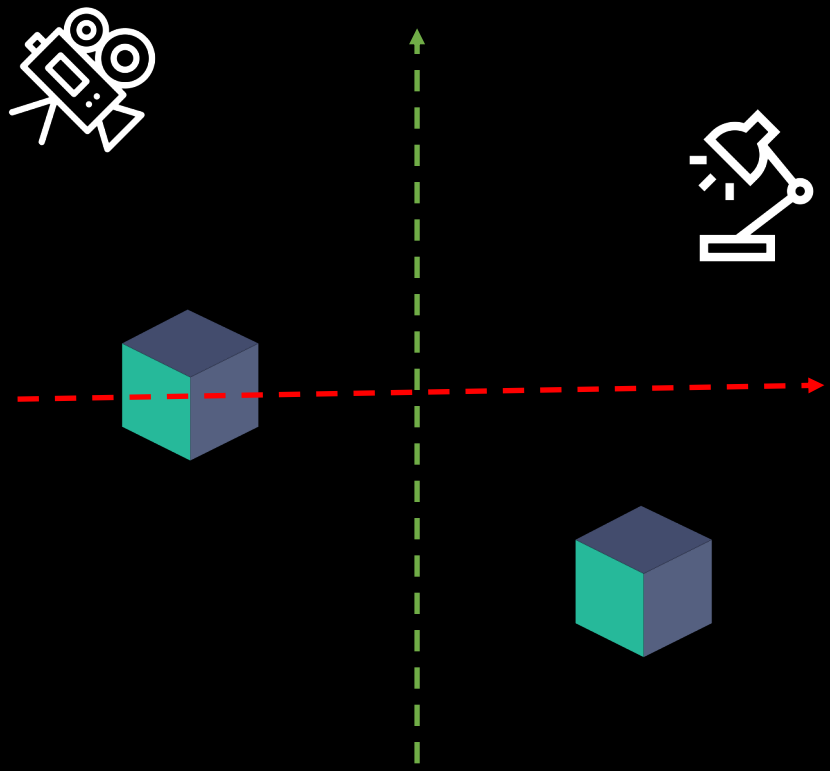


Матрица2 * Матрица1

Пространство объектов



Локальное пространство

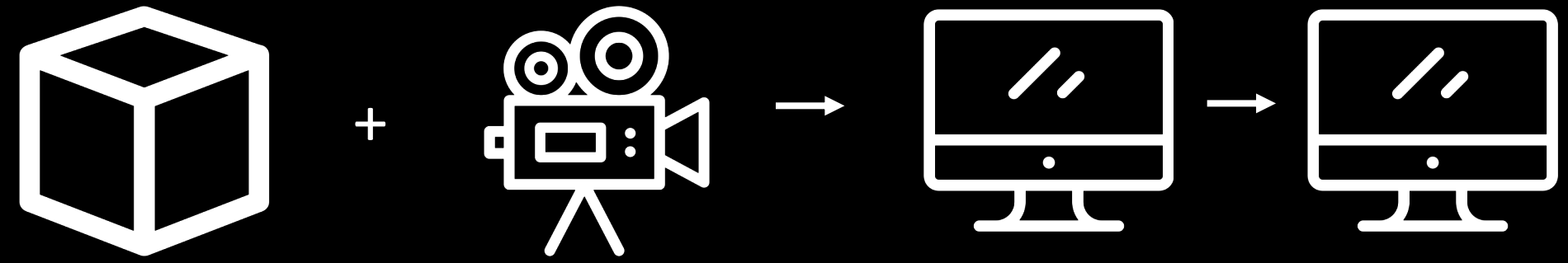


Мировое пространство

Мировая матрица = Мировая родителя * локальная объекта

Модель, вид, проекция

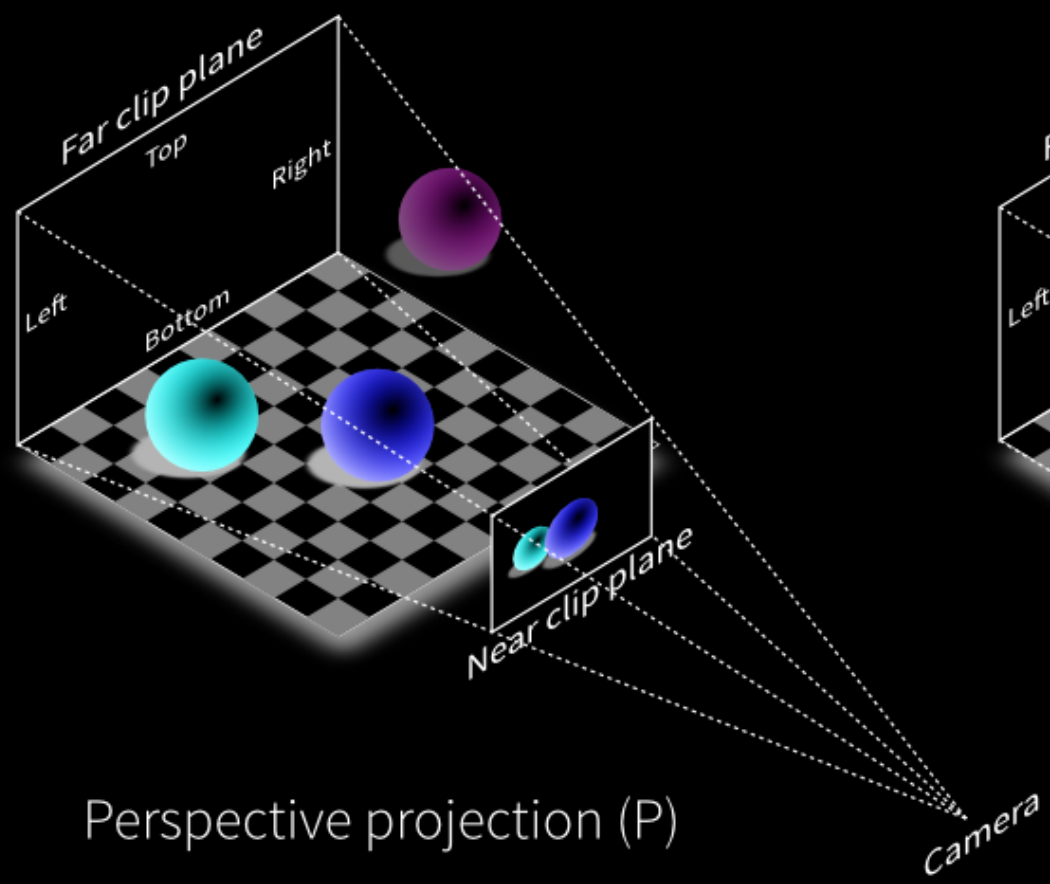
Экранные координаты = мировая матрица объекта * матрица вида * матрица проекции * (ширина и высота экрана)



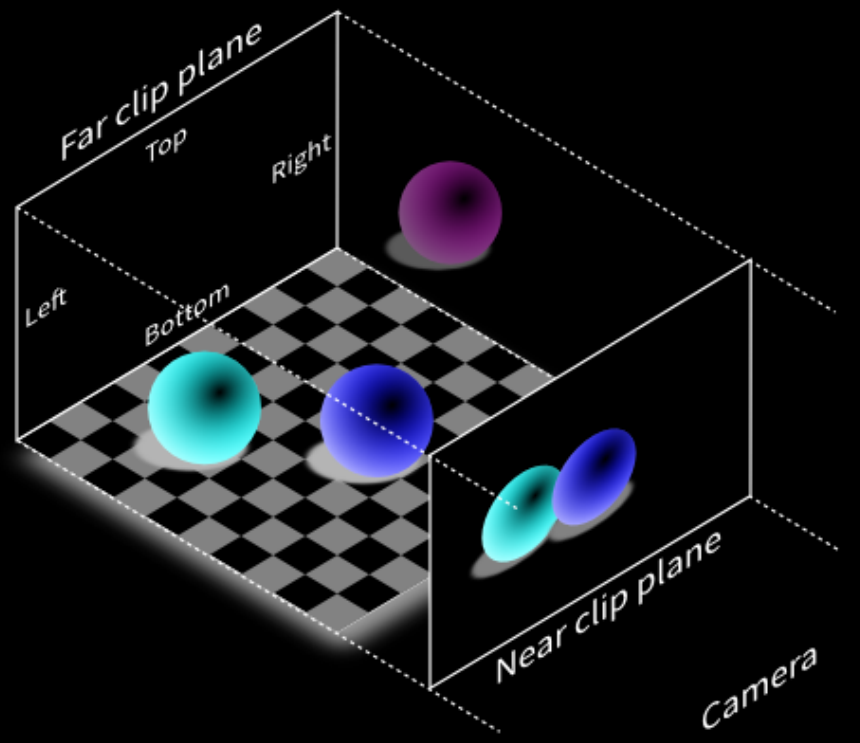
Камера

```
// создаем камеру
const camera = gl.cre
// смотрим на объек
camera.lookAt(obje
// масштабируем в
camera.zoom(2);
// анимируем вращ
camera.animate({
      type: 'loop'
      property: 'ro
      time: '10s'
});
```

Камера



Perspective projection (P)



Orthographic projection (O)

Камера

```
interface Camera {
    view: Matrix4;
    projection: Matrix4;
}
```

```
interface PerspectiveProps {
    nearPlane: float;
    farPlane: float;
    fieldOfView: float;
    aspectRatio: float;
}
```

```
camera.view = invert(camera.model)
```

```
camera.projection = calculatePerspective(perspectiveProps)
```

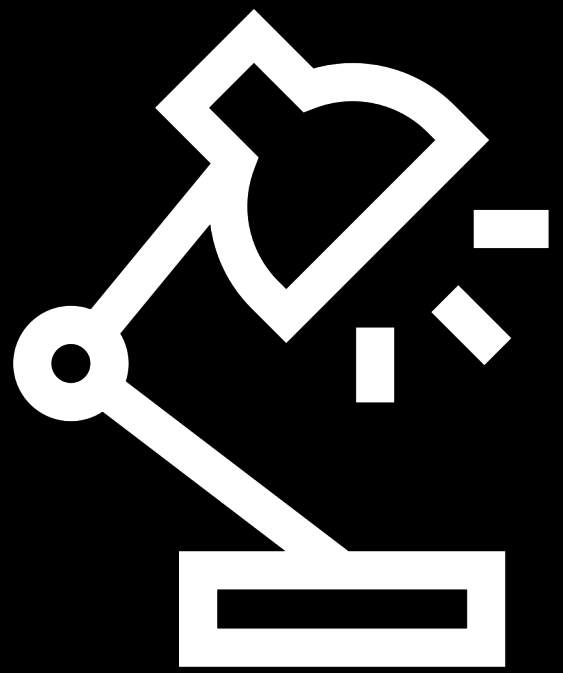
Черный экран

```
gl_Position = projection * view * model * vec4(position, 1.0);
```

```
gl_Position = vec4(position, 1.0); // -1.0 <= position >= 1.0
```

Источник света

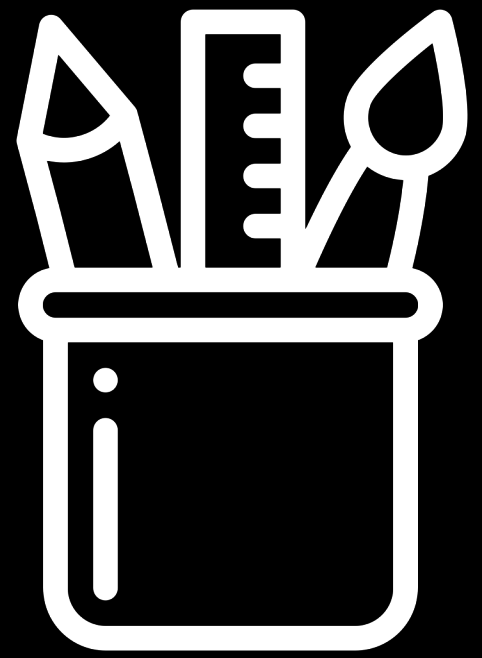
```
interface Light {
    position: Vec3;
    color: Vec3;
    power: float;
}
```



Материал

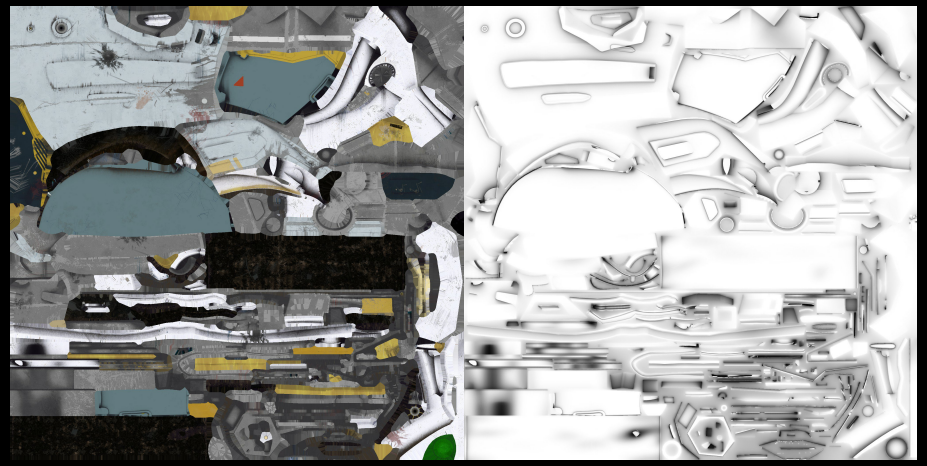
gltf.materials

```
interface Material {
        color: Vec4 | Texture;
        metallic: float | Texture;
        roughness: float | Texture;
        emissive: Vec4 | Texture;
        ao: float | Texture;
        normal: Vec3 | Texture;
}
```



Текстуры

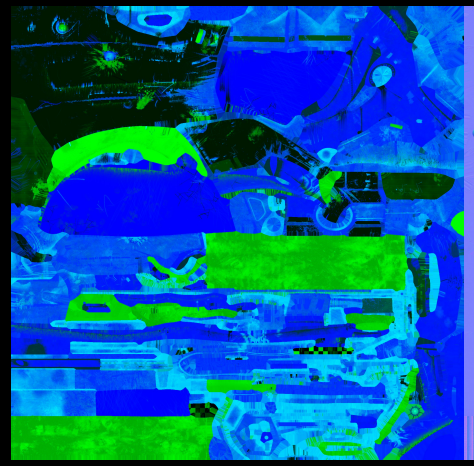
gltf.textures -> gltf.images и gltf.samplers



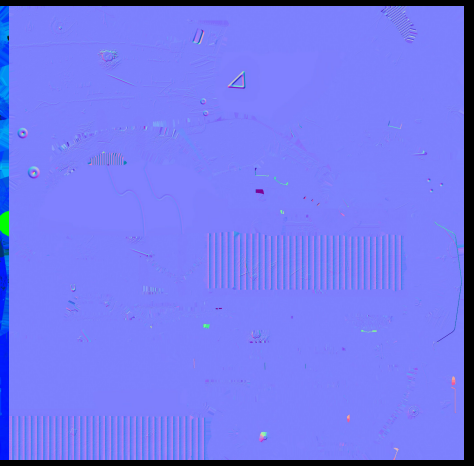
Альбедо

Затенение

Излучение



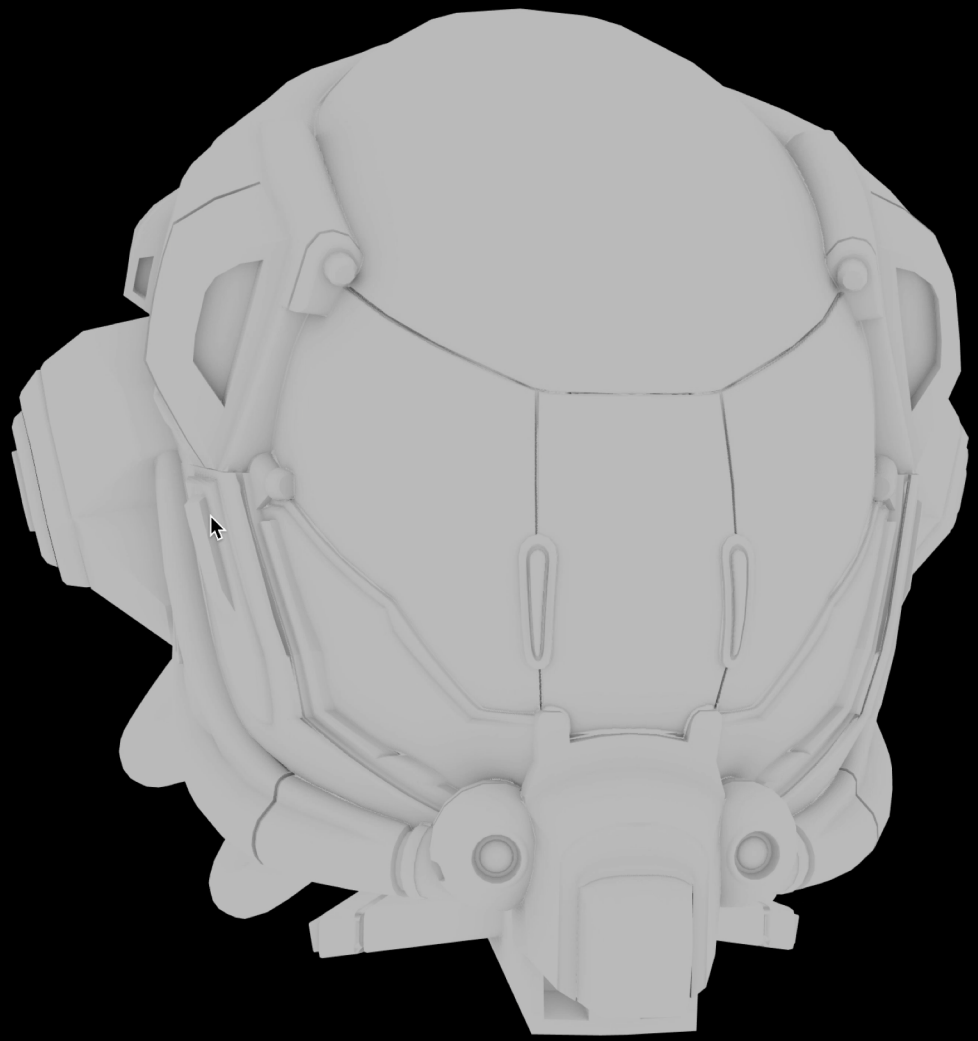
Шероховатость и металличность



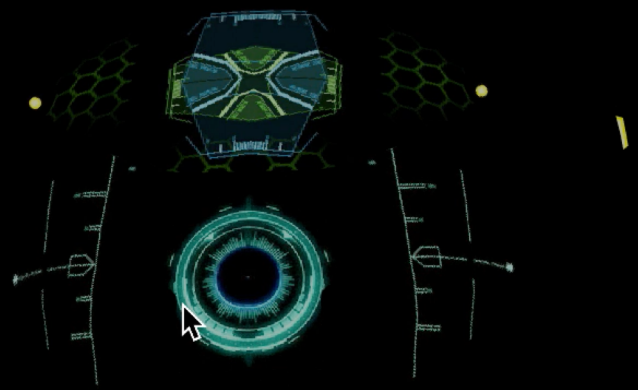
Нормали

Основной цвет

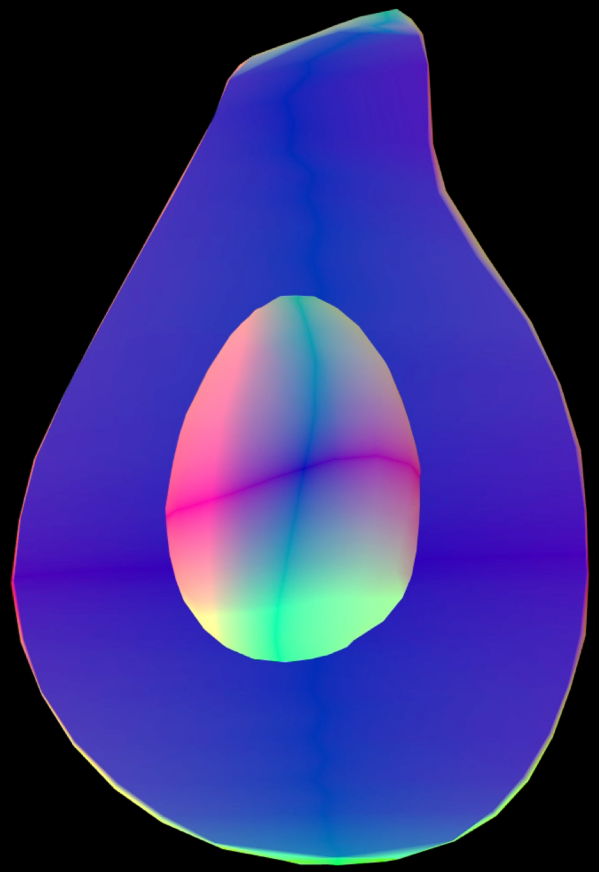
Окружающее затенение



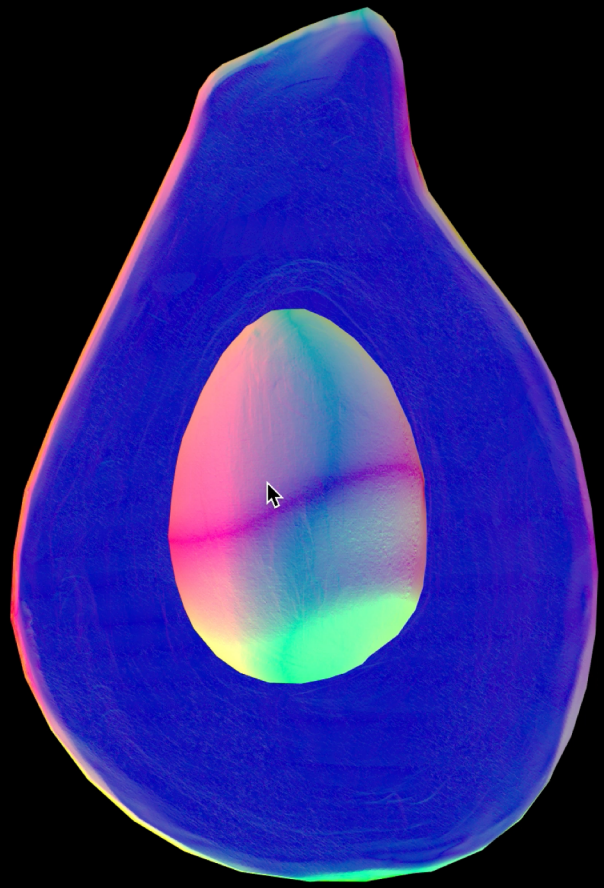
Излучение



Карта нормалей

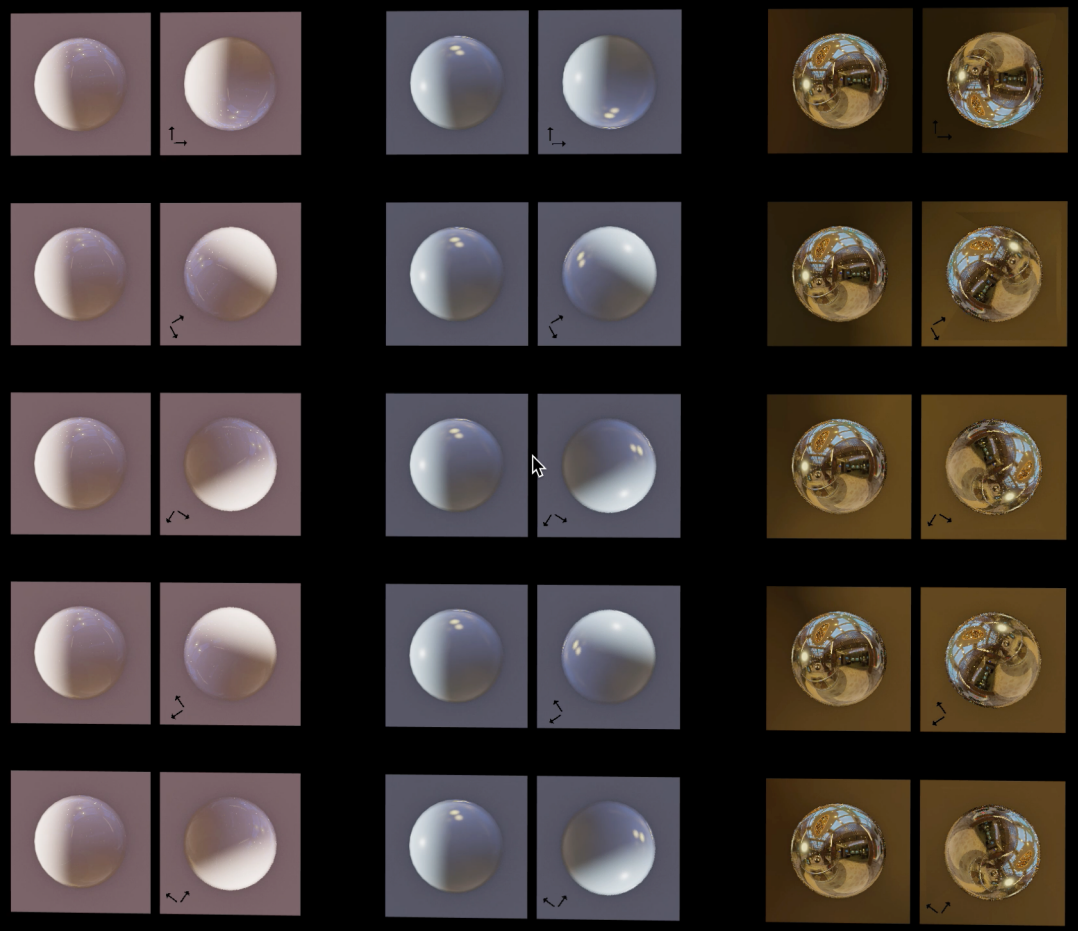


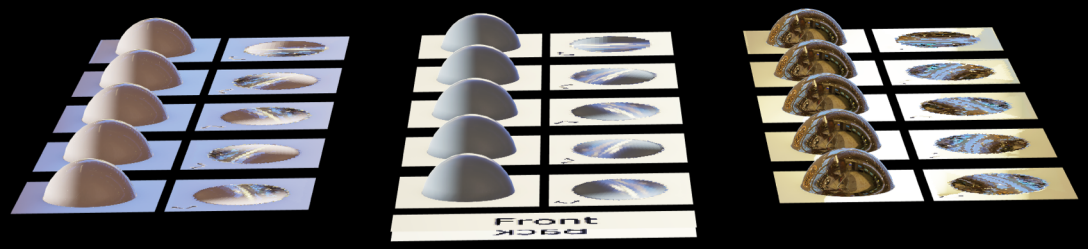
Без рельефа



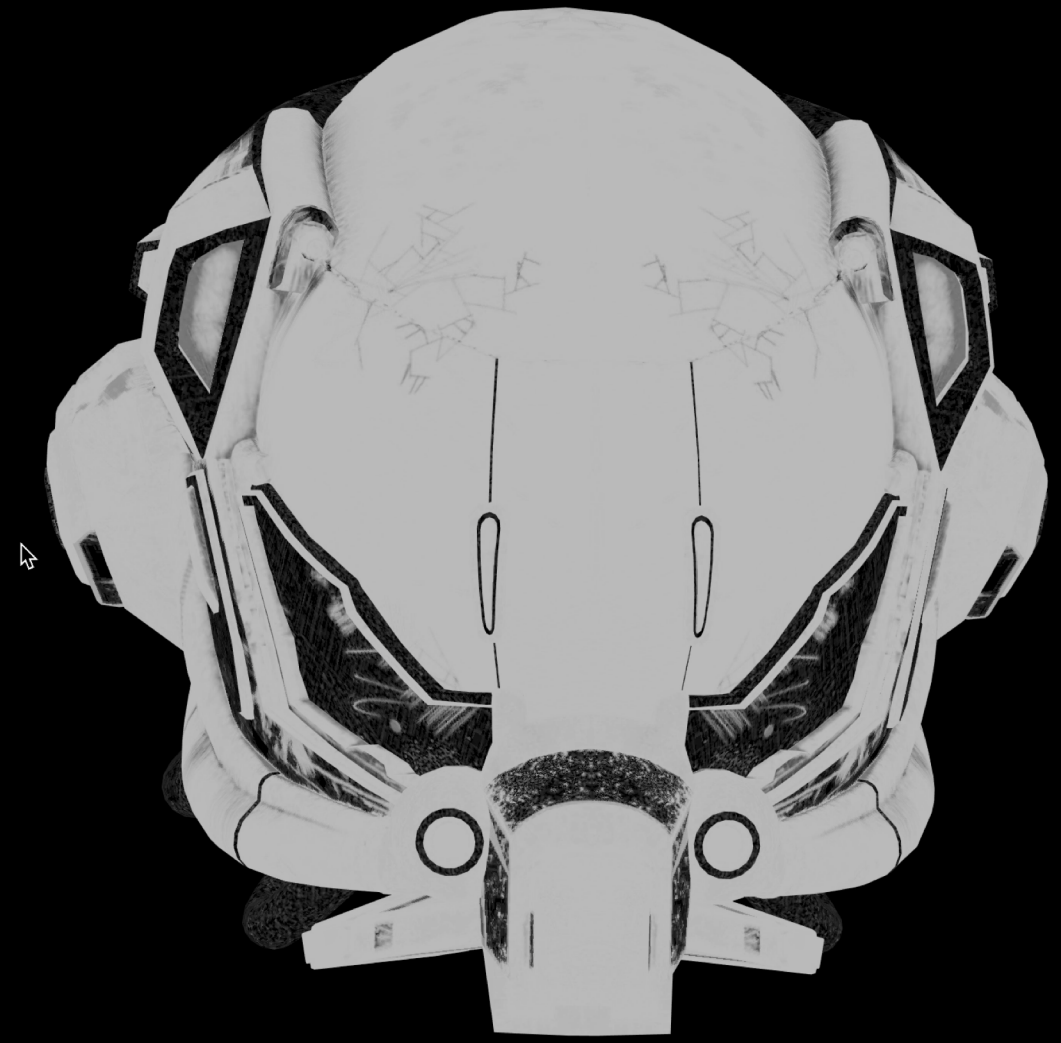
С рельефом

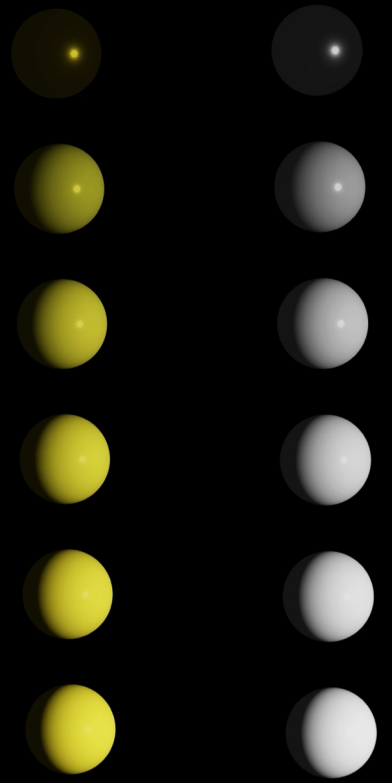
Карта нормалей



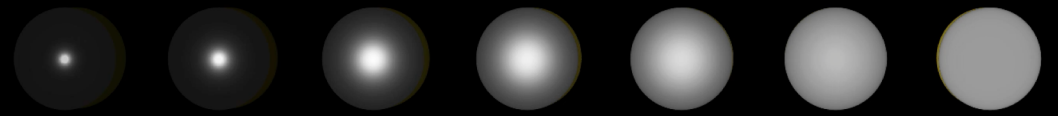


Металлическая поверхность





Шероховатость



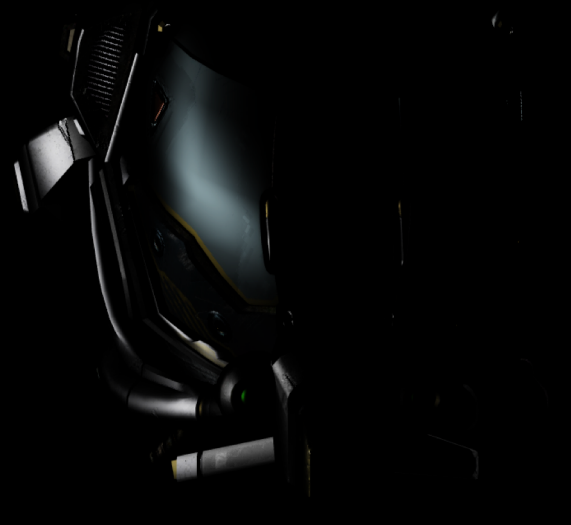
Гладкий

Грубый

Компоненты освещения

Свет окружающей среды

Рассеянный свет



Отраженный свет

Модели освещения

Рассеянный свет:

- Ламберт – рассеянный свет
- Орен-Наяр – рассеянный свет

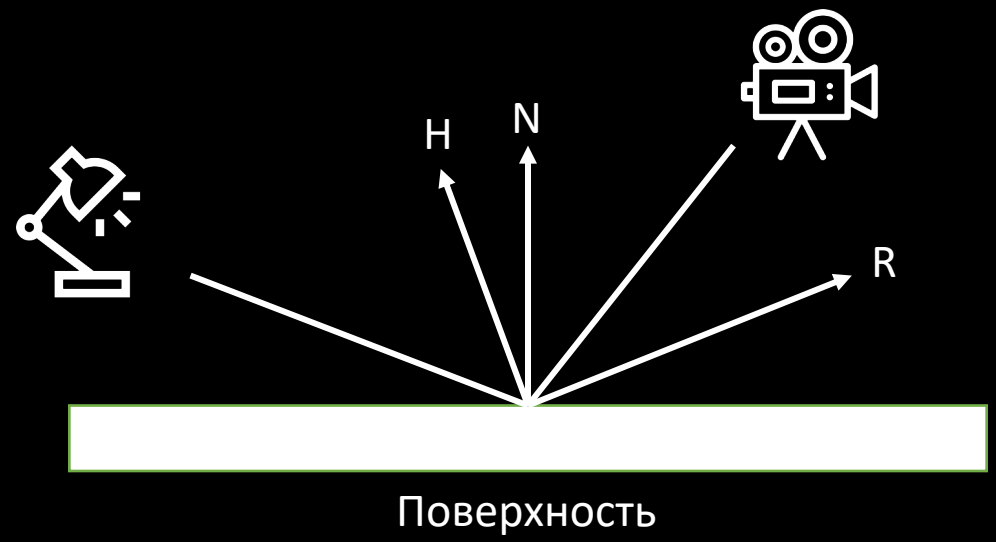
Отраженный свет:

- Фонг
- Изотропный и Анизотропный Уорда

Физически точное освещение – общее название техник, реализующих освещение приближенное к реальному

Входные данные

- V – направление камеры
- L – направление света
- N – нормаль меша
- H – вектор точно посередине между направлением камеры и направлением света
- R – отражение света относительно нормали



Физически-точный рендеринг

Модель отражения Фонга

Физически-точный рендеринг

- Walt Disney Animation Studios

https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf

- Unreal Engine 4
  https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

Цветовое пространство

WebGL

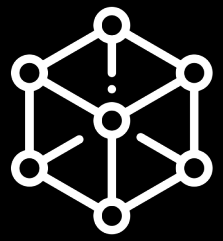
255 * 0.5 = 128

Монитор

255 * 0.5 = 187

Шейдеры

Шейдер – это программа для GPU



Вершинный

Фрагментный

OpenGL

Геометрический

Тесселяция

Вершинный шейдер и геометрия

```
// входные данные
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 uv;

// выходные данные
out vec3 outNormal;
out vec2 outUV;
outUV = uv;
outNormal = normal;

gl_Position = projection * view * model * vec4(position, 1.0);
```

Фрагментный шейдер и материал

```
layout (location = 0) out vec4 color;

uniform Material {
  vec3 baseColor;
}

vec3 L = normalize(lightPosition - position);
float NL = dot(N, L);

color = vec4(baseColor * NL, 1.0);
```

Про шейдер

```
// GLSL
vec3 pos = vec3(1.0);
vec2 uv = vec2(0.0);
vec3 sum = pos + uv;

// JS
gl.getShaderInfoLog(shader)
```

ERROR: 0:52: '+' : wrong operand types - no operation '+' exists that takes a left-hand operand of type 'in highp 2-component vector of float' and a right operand of type 'in highp 3-component vector of float' (or there is no acceptable conversion)

Конвейер рендеринга

```
for (const mesh of scene) {
        // привязка шейдеров
        gl.useProgram(mesh.program);
        // привязка геометрии
        gl.bindVertexArray(mesh.geometry.buffer);
        // привязка матриц MVP
        gl.bindBufferBase(gl.UNIFORM_BUFFER, 0, mesh.geometry.matrices);
        // привязка материала
        gl.bindBufferBase(gl.UNIFORM_BUFFER, 1, mesh.material);
        // рисуем
        gl.drawElements(mesh.mode, mesh.geometry.length, mesh.geometry.type, 0);
}
```

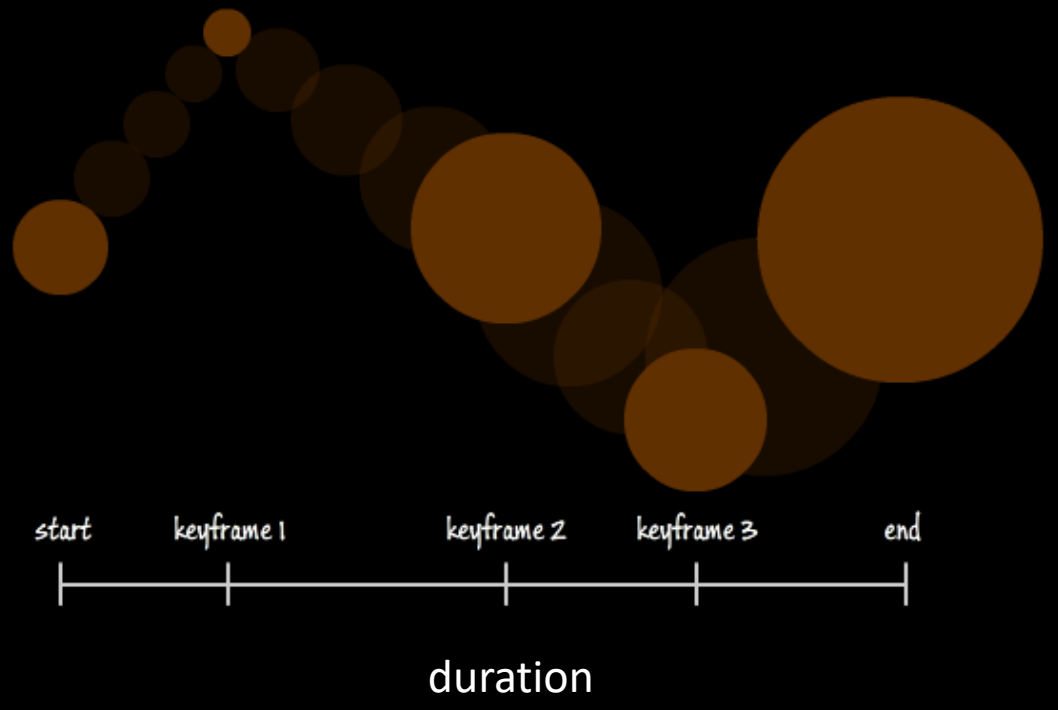
Конвейер рендеринга

Анимация

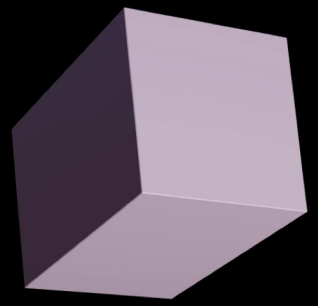
gltf.animations

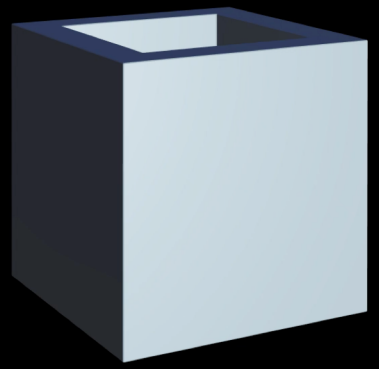
```
interface AnimationObject {
  interpolation: "LINEAR" | "STEP";
  keys: Array<AnimationKey>;
  meshes: Array<Mesh>;
  type: "rotation" | "translation" | "scale";
}

interface AnimationKey {
  time: number;
  value: Vec3;
}
```



Анимация



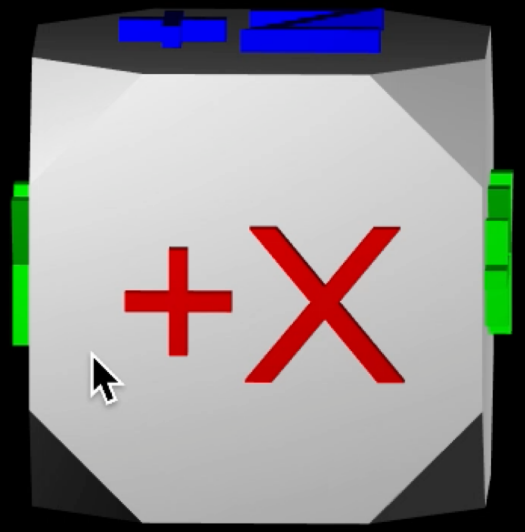


AnimationObject - 2
AnimationKey - 6

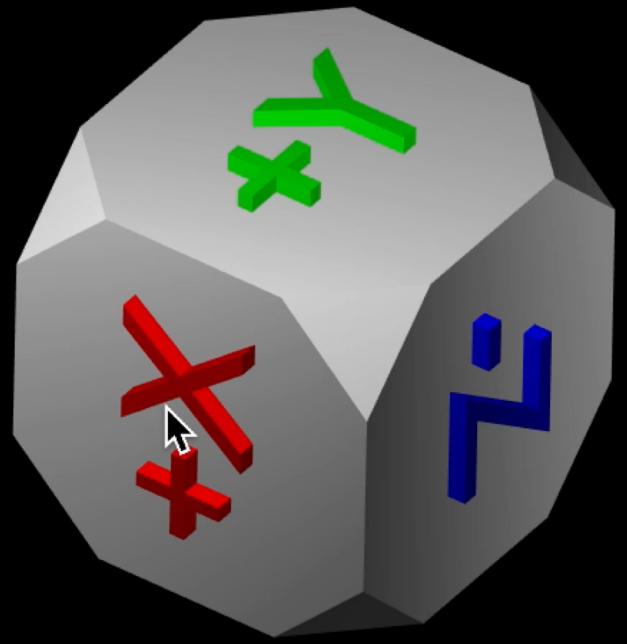
Вращение

Алгоритмы	Произвольное вращение	Независимо от пути	Неограниченное вращение
Turntable	⊘	✔	✔
Tumbler	✔	⊘	✔
Trackball	✔	✔	⊘

Вращение



Заблокированная ось



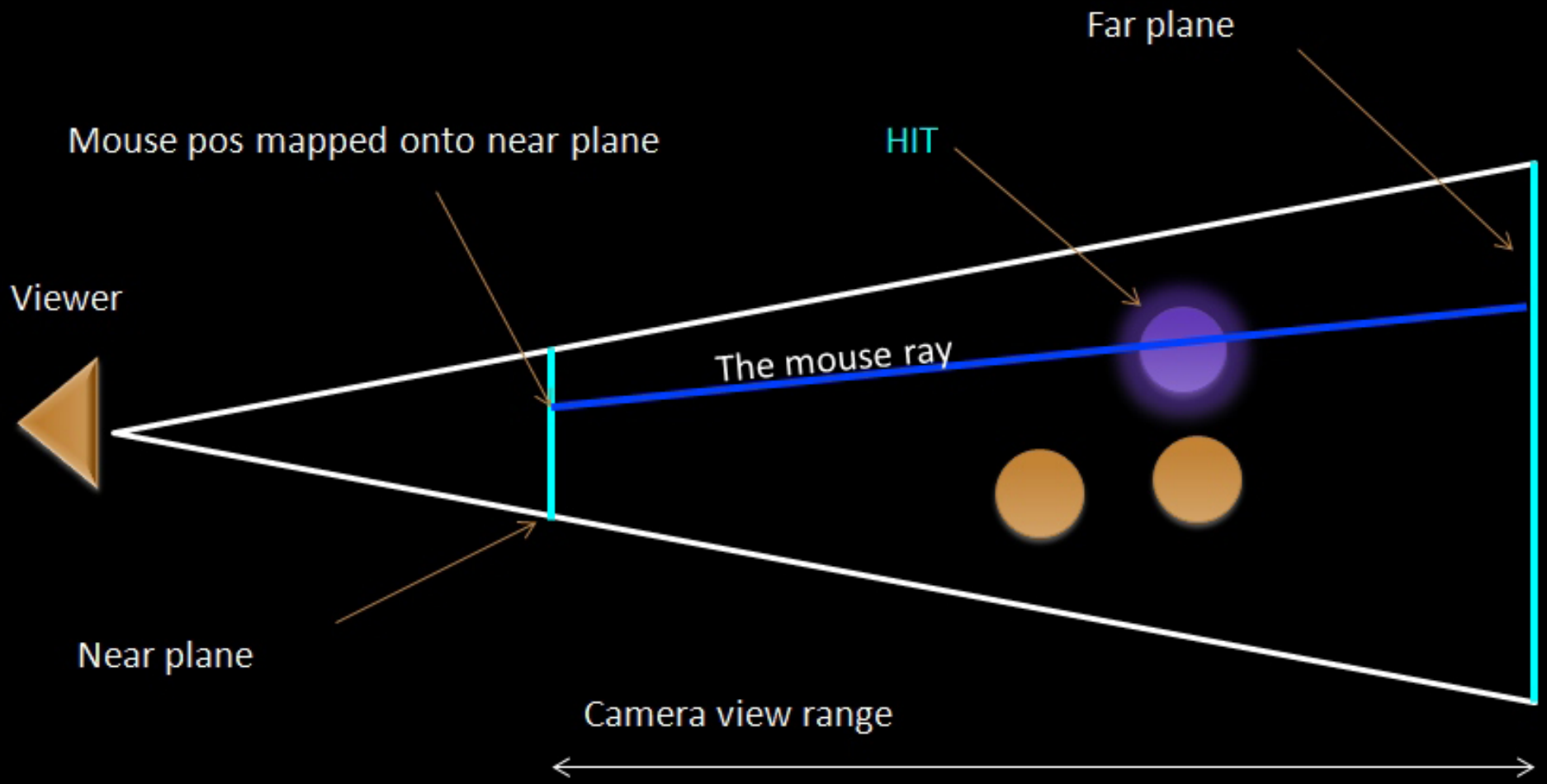
Произвольное вращение

Масштабирование сцены

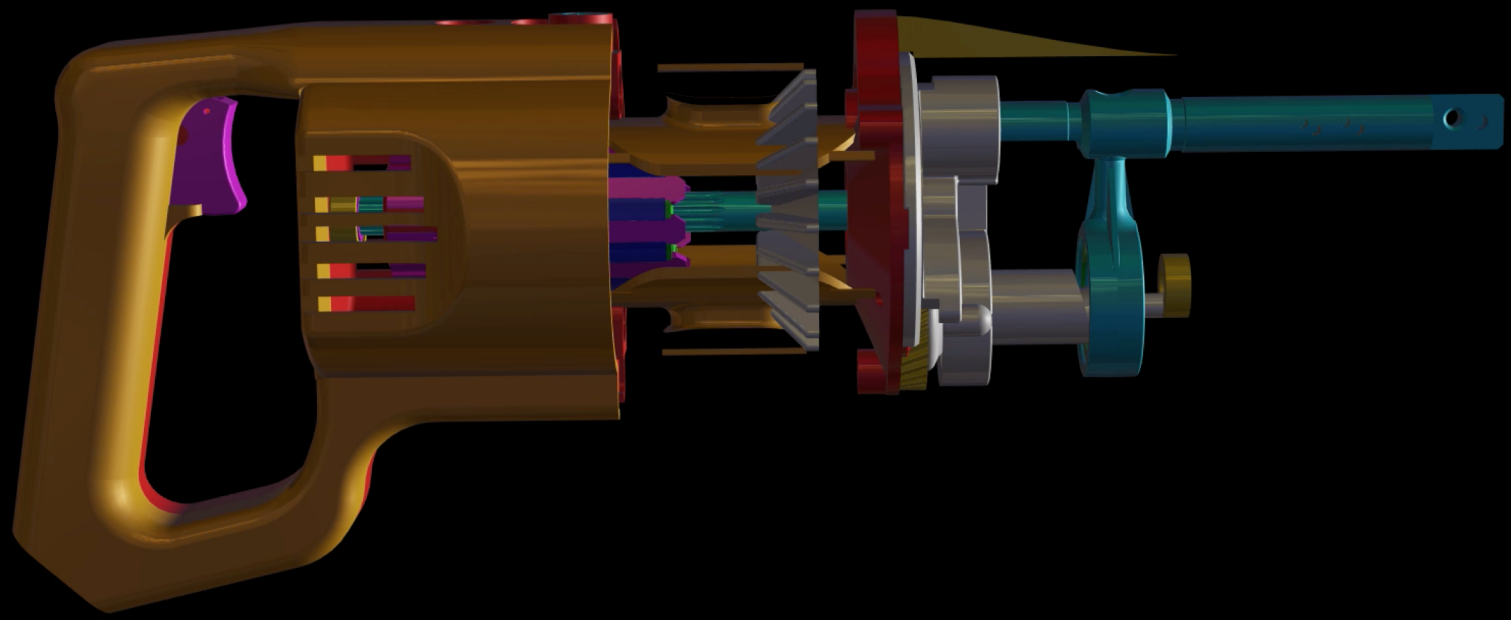
Угол обзора 15°,
сдвиг камеры по Z

Угол обзора 60°

Выбор объекта на сцене



Выбор объекта на сцене

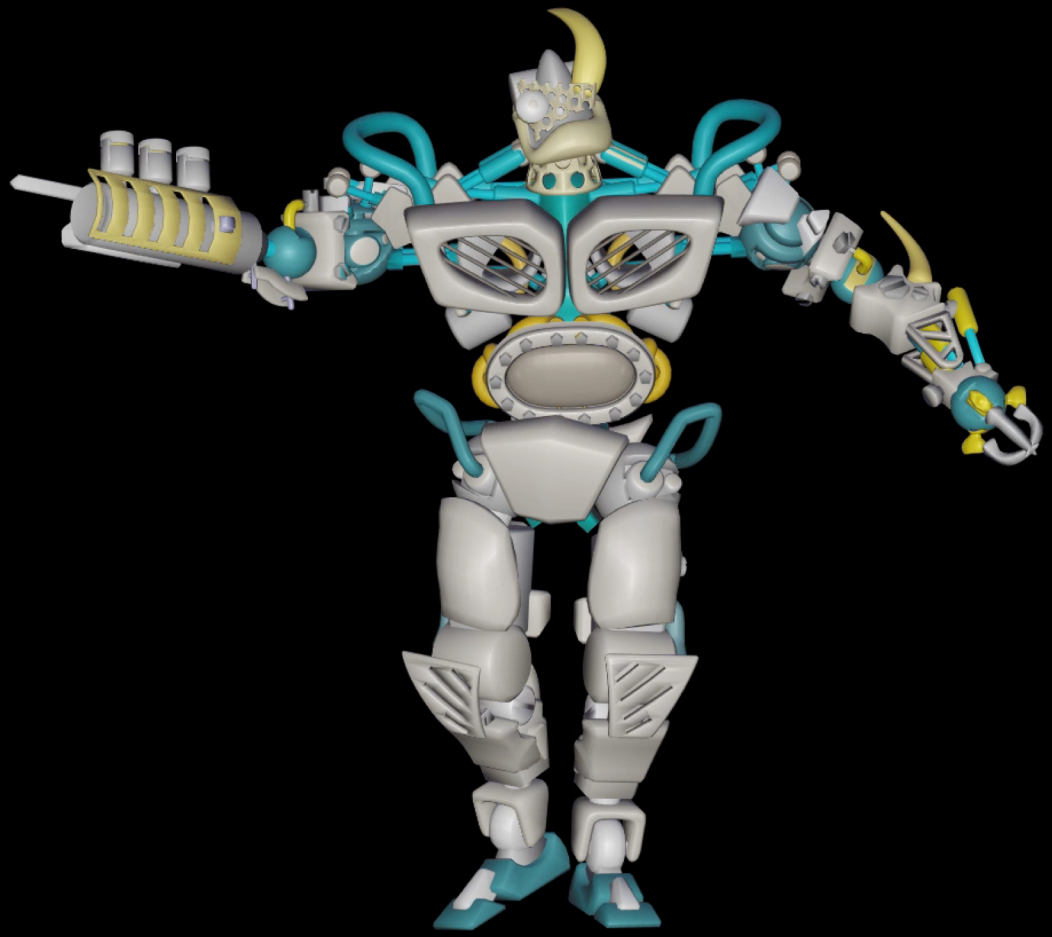


Выводы

- Используйте спецификацию GLTF, способствуйте ее развитию
- Попробуйте сделать что-то в 3D прямо сегодня

В каких направлениях можно развиваться?

Скелетная анимация



Морфинг

Физически точный рендеринг

Пост обработка

Система частиц

Redcube.js

Репозиторий:
https://github.com/Reon90/redcube

Демо:
https://reon90.github.io/redcube/