Компьютерная графика
Современные технологии компьютерной графики и рендеринга

Лекция 4

02.04.02 ФИИТ

Разработка мобильных приложений и компьютерных игр

2025-2026

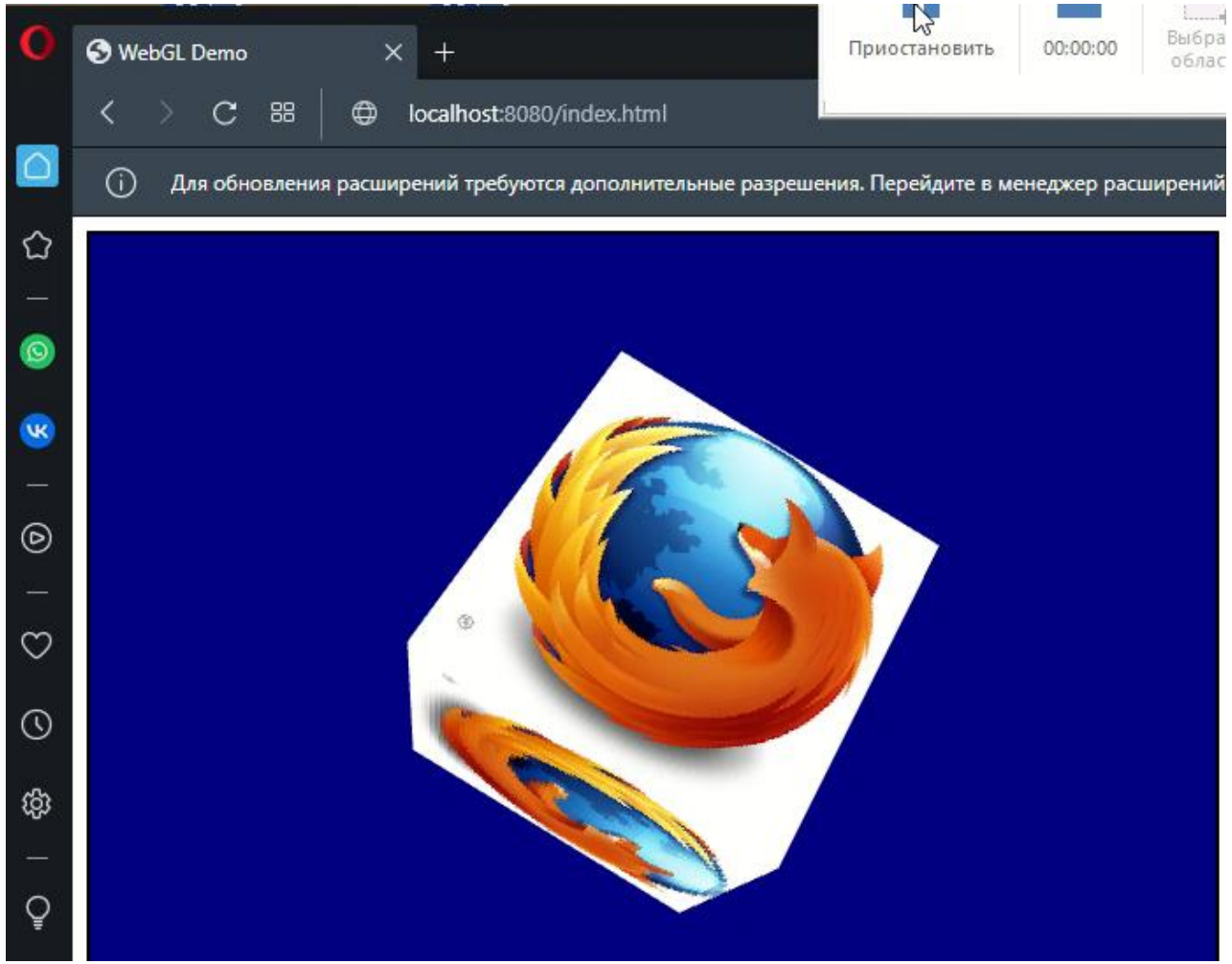
Где мы были и куда идем?

Лекции 1-2
- Матрицы преобразований, системы координат
- Буферы, атрибуты, вершинные и фрагментные шейдеры
- Рисование простых примитивов

Лекция 3
- Модели освещения (Ламберт, Фонг, Блинн-Фонг)
- Нормали и матрица нормалей
- Материалы и их взаимодействие со светом

Лекция 4
- Текстуры — добавляем детализацию без усложнения геометрии
- Прозрачность и смешивание
- Форматы 3D-моделей (OBJ, MTL, COLLADA)
- Загрузка готовых моделей вместо ручного описания вершин

Текстуры в WebGL. Что хотим получить



Что передаётся в шейдеры

```
const programInfo = {
    program: shaderProgram,
    attribLocations: {
      vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
      textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),
    },
    uniformLocations: {
      projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
      modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
      uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),
    }
  };
```

Создание переменной — идентификатора текстуры

```
//глобальная переменная

var texture = gl.createTexture();
```

Загрузка текстуры

Для установки изображения в качестве текстуры нам нужен элемент
var image = new Image();

Изображение, установленное для данного элемента, и будет устанавливаться в качестве текстуры.

Два способа:

- заранее определить в структуре DOM веб-страницы элемент **img** и с помощью его атрибута **src** установить какое-либо изображение
- динамически в коде javascript создать данный элемент

Если динамически

Поскольку текстура **не сразу загружается**, то используем обработку события **onload**:

```
function setTextures(){
   texture = gl.createTexture();
   gl.bindTexture(gl.TEXTURE_2D, texture);

   var image = new Image();
   image.crossOrigin = "anonymous";//уже не помогает)

   image.onload = function() {
         handleTextureLoaded(image, texture);
         setupWebGL();
         draw();
    }

    image.src = "brick_045.jpg";
    shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
    gl.uniform1i(shaderProgram.samplerUniform, 0);
}
```

Cross-Origin Resource Sharing (CORS)

Важно помнить, что загрузка текстур следует правилам кросс-доменности, что означает, что вы можете загружать текстуры только с сайтов, для которых ваш контент является CORS доверенным.

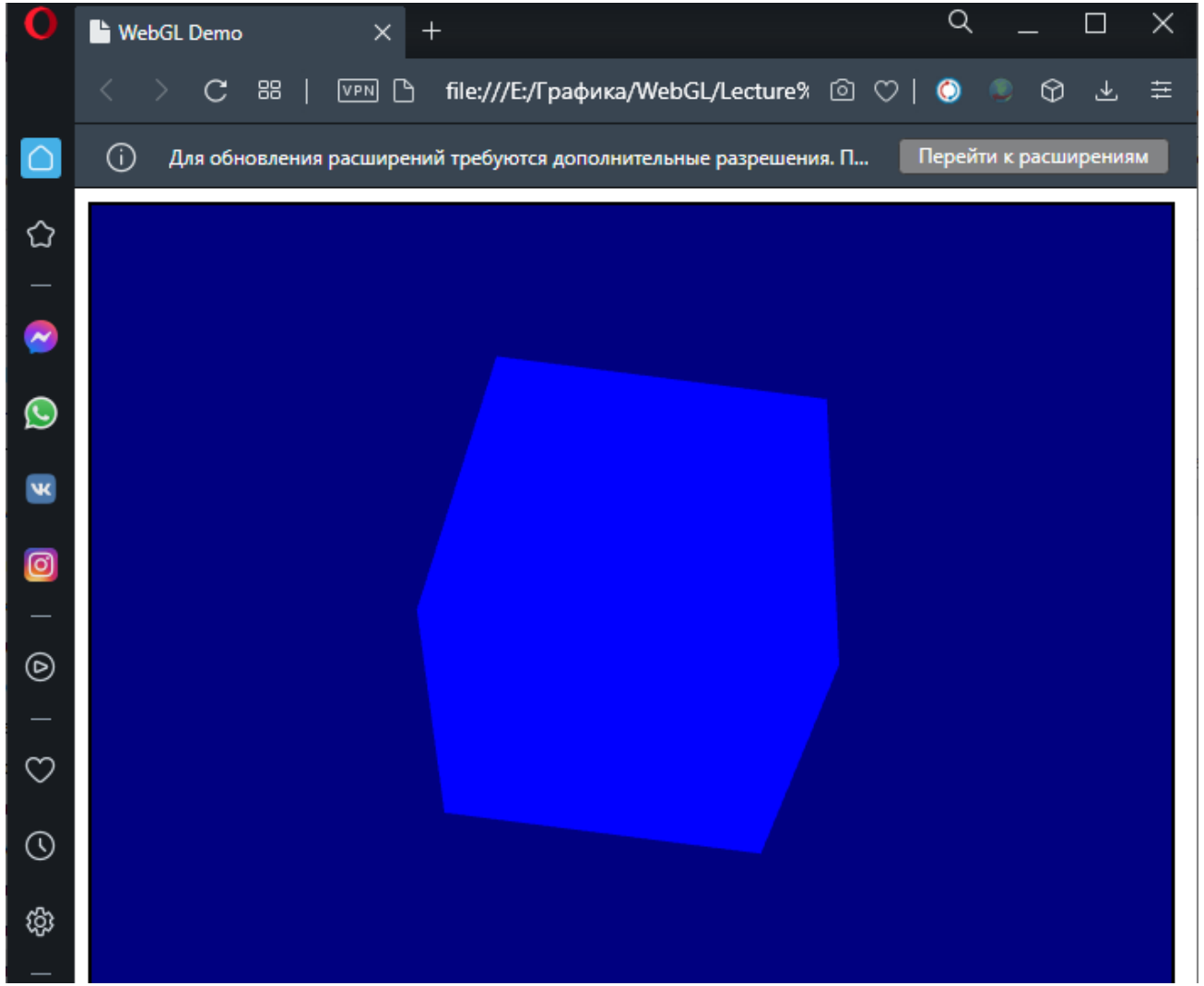
- Браузеры блокируют загрузку текстур с других доменов по соображениям безопасности
- Изображение должно быть с того же домена или сервер должен разрешать CORS
- Чтобы работало везде, нужно запустить локальный сервер, например:

```
npx http-server
# или
python -m http.server
```

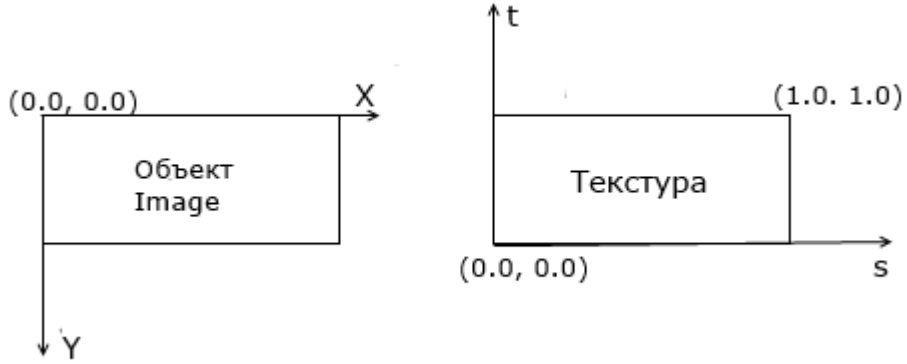
А пока текстура загружается…

```
const pixel = new Uint8Array([0, 0, 255, 255]);  // непрозрачный синий

const level = 0;
const internalFormat = gl.RGBA;
const width = 1;
const height = 1;
const border = 0;
const srcFormat = gl.RGBA;
const srcType = gl.UNSIGNED_BYTE;

gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, width, height, border, srcFormat, srcType, pixel);
```

Заглушка на время загрузки

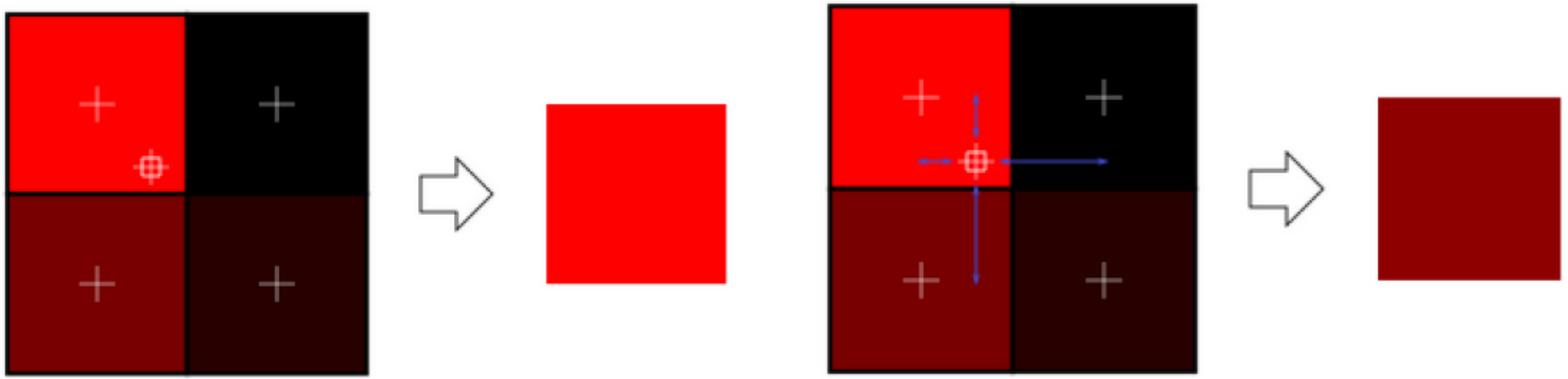


Настройка параметров текстурирования



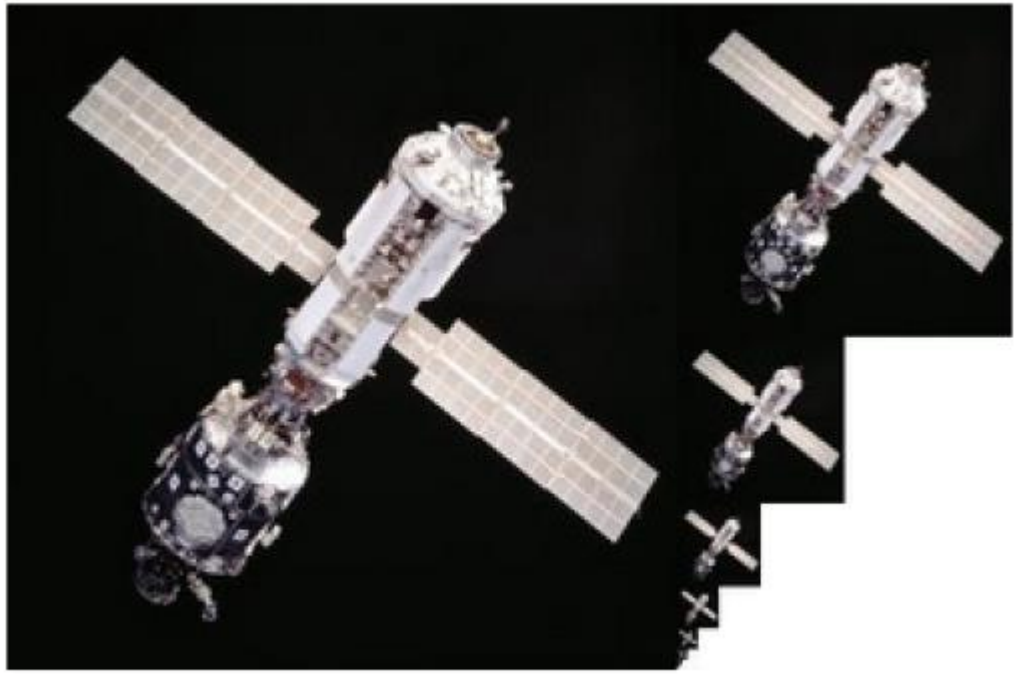

```
function handleTextureLoaded(image, texture) {

   gl.bindTexture(gl.TEXTURE_2D, texture);

   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  //указывает далее идущему методу gl.texImage2D(), как текстура должна позиционироваться. Так, в данном случае мы
     передаем в качестве параметра значение  gl.UNPACK_FLIP_Y_WEBGL - этот параметр указывает методу
     gl.texImage2D(), что изображение надо перевернуть относительно горизонтальной оси.

  // Загружаем изображение в текстуру
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

   // Устанавливаем фильтры
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
}
```

Фильтрации текстур



- NEAREST (пикселизация) — выбирает ближайший тексель. Быстро, но дает квадратный эффект при увеличении
- LINEAR (билинейная) — интерполирует между 4 ближайшими текселями. Медленнее, но гладко
- Для уменьшения также используется MIP-фильтрация

MIPMAP (пирамида текстур)



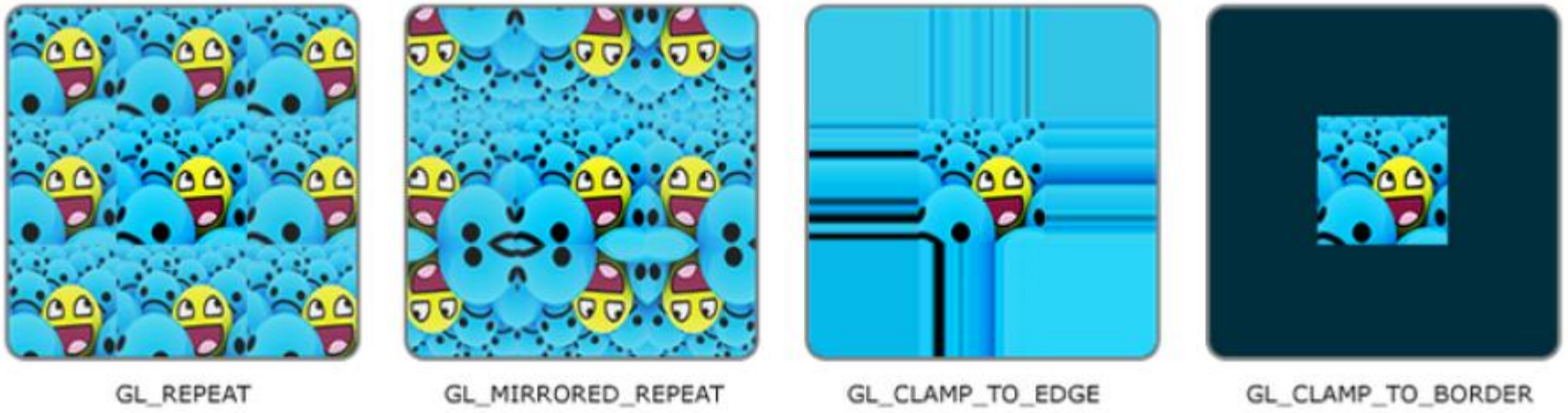
Варианты фильтрации текстур

```
 // У WebGL1 разные требования к изображениям, имеющим размер степени 2,
// и к изображениям, не имеющим размер степени 2

if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
      // Размер соответствует степени 2. Создаем MIP'ы.
      gl.generateMipmap(gl.TEXTURE_2D);
 } else {
      // Размер не соответствует степени 2.
      // Отключаем MIP’ы и повторение и устанавливаем натяжение по краям

      // также разрешено gl.NEAREST вместо gl.LINEAR, но не mipmap.
     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

     // Не допускаем повторения по s-координате.
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);

    // Не допускаем повторения по t-координате.
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
}
```

Режимы повтора текстуры

Если текстурные координаты выйдут за промежуток 0-1

GL_REPEAT: по умолчанию. Повторяет текстуру
GL_MIRRORED_REPEAT: Похож на GL_REPEAT, но отражается
GL_CLAMP_TP_EDGE: привязывает координаты между 0 и 1. Выход за пределы координат будут привязаны к границам
GL_CLAMP_TO_BORDER: Координаты, выходящие за пределы, будут цвета границы

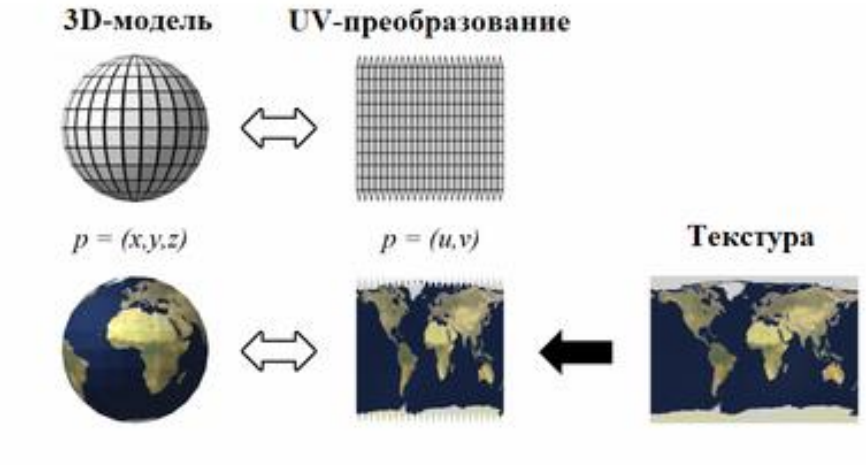


UV-преобразование или развёртка

UV-преобразование или развёртка в трёхмерной графике (англ. UV map)

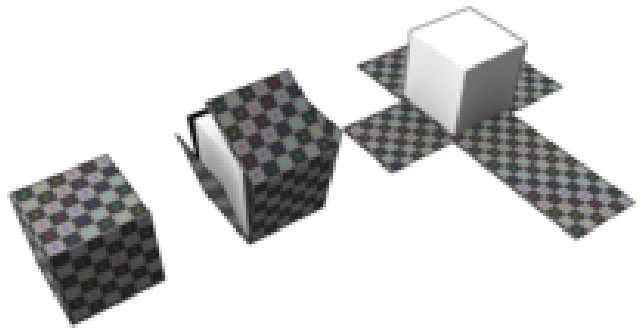
UV-развертка— соответствие между координатами на поверхности 3D-объекта (X, Y, Z) и координатами на текстуре (U, V)

- U и V обычно изменяются от 0 до 1
- Каждая вершина получает пару (U, V)
- Растеризатор интерполирует UV для каждого фрагмента



Развёртка может строиться как вручную, так и автоматически — например, в 3Ds Studio MAX есть несколько алгоритмов автоматического развёртывания модели

Примеры развёрток



Пример наложения текстуры на куб.
Текстура имеет вид плоской развёртки

Шахматная текстура на сфере:
слева — общая матрица на всю сферу,
справа — развёртка с равнопромежуточной проекцией

Равнопромежуточная проекция — пример наложения двухмерной текстуры (координаты U, V) на трёхмерный глобус (координаты X, Y, Z)

Пример координат текстуры

```
const textureCoordinates = [
   // Front
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
   // Back
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
   // Top
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
   // Bottom
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
   // Right
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
   // Left
   0.0,  0.0,          1.0,  0.0,          1.0,  1.0,          0.0,  1.0,
 ];
```

Координаты текстуры лежат в промежутке между 0.0 и 1.0.
Размерность текстуры нормализуется в пределах между 0.0 и 1.0, независимо от реального размера изображения

Вершинный шейдер с текстурированием

```
#version 300 es
in vec4 aVertexPosition;
in vec2 aTextureCoord;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;

out vec2 vTextureCoord;

void main() {
   gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
   vTextureCoord = aTextureCoord;
}
```

Фрагментный шейдер с текстурированием

```
#version 300 es
precision highp float;

in vec2 vTextureCoord;
uniform sampler2D uSampler;
out vec4 fragColor;

void main() {
   fragColor = texture(uSampler, vTextureCoord);
}
```

Настройка атрибута текстурных координат

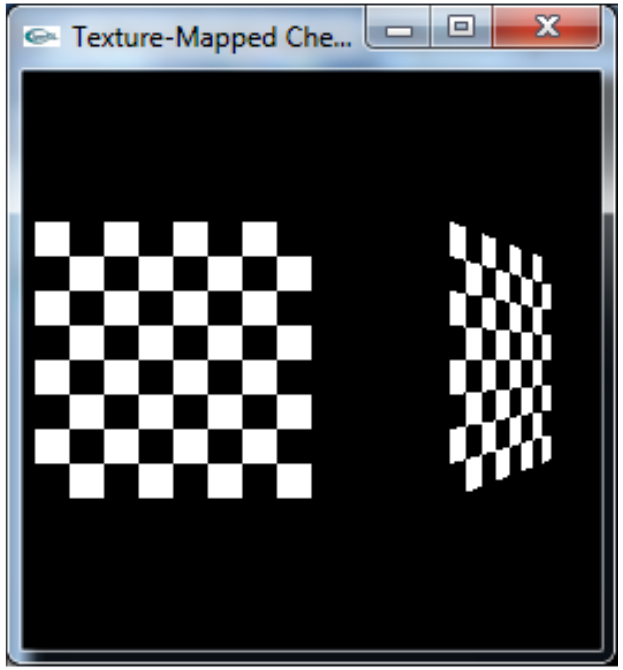
```
const num = 2; // каждая текстурная координата состоит из 2 значений (U, V)
const type = gl.FLOAT; // данные в буфере имеют тип 32-bit float
const normalize = false; // не нормализуем
const stride = 0; // сколько байт между одним набором данных и следующим
const offset = 0; // стартовая позиция в байтах внутри набора данных

gl.bindBuffer(gl.ARRAY_BUFFER, buffers.textureCoord);

gl.vertexAttribPointer(programInfo.attribLocations.textureCoord, num, type, normalize, stride, offset);

gl.enableVertexAttribArray(programInfo.attribLocations.textureCoord);
```

Активные текстурные юниты

```
// WebGL имеет минимум 8 текстурных регистров; первый из них gl.TEXTURE0


// Указываем WebGL, что мы используем текстурный регистр 0
gl.activeTexture(gl.TEXTURE0);

// Связываем текстуру с регистром 0 (юнитом 0)
gl.bindTexture(gl.TEXTURE_2D, texture);

// Указываем шейдеру, что мы связали текстуру с текстурным регистром 0
// Передаем номер юнита в шейдер
gl.uniform1i(programInfo.uniformLocations.uSampler, 0);
```

Результат наложения текстуры

Программная генерация текстур




```
checkImage [ checkImageHeight ] [ checkImageWidth ] [ 4 ] ;


for ( i =0; i<checkImageHeight ; i++) {
    for ( j =0; j<checkImageWidth ; j++) {
            c=((( i&0x8 )==0)^(( j&0x8 )==0))255;
            checkImage [ i ] [ j ] [ 0 ]=( GLubyte ) c ;
            checkImage [ i ] [ j ] [ 1 ]=( GLubyte ) c ;
            checkImage [ i ] [ j ] [ 2 ]=( GLubyte ) c ;
            checkImage [ i ] [ j ] [ 3 ]=( GLubyte ) 255 ;
            }
     }
```

Прозрачность и смешивание текстур

Имитация смешивания — Отбрасывание фрагментов

Реальное смешивание

Рендер полупрозрачных текстур

Имитация смешивания — Отбрасывание фрагментов

Отбрасываем фрагменты, содержащие прозрачные части текстуры, не сохраняя их в буфере цвета

Как учесть прозрачность?

gl.TexImage2D(GL_TEXTURE_2D, 0, **GL_RGBA**, width, height, 0, **GL_RGBA**, GL_UNSIGNED_BYTE, data);

Во фрагментом шейдере выборка в вектор с **4мя компонентами**, чтобы не остаться только с RGB значениями:

```
void main() {
   // FragColor = vec4(vec3(texture(texture1, TexCoords)), 1.0);
   FragColor = texture(texture1, TexCoords);
}
```

Отбрасывание фрагментов (discard)

```
// Во фрагментном шейдере можно отбросить пиксели с alpha < порога

#version 330 es
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main() {
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard; // пиксель не будет записан в буфер
    FragColor = texColor;
}
```

Как избежать артефакта рамки?

Для избежания артефакта появления полупрозрачной цветной рамки вокруг выведенной текстуры при использовании текстур с прозрачностью нужно параметр повтора установить в GL_CLAMP_TO_EDGE.

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Режимы смешивания

Для рендера изображений с объектами, имеющими разную степень непрозрачности мы должны включить режим смешивания

gl.Enable(GL_BLEND);

Как работает смешивание

Смешивание WebGL выполняется по следующей формуле

$$C_{result} = C_{source} * F_{source} + C_{destination} * F_{destination}$$

где

C_source – вектор цвета источника (нового фрагмента). Это значение цвета, полученное из текстуры

F_source – множитель источника. Задает степень влияния альфа-компоненты на цвет источника

C_destination – вектор цвета приемника. Это значение цвета, хранимое на данный момент в буфере цвета

F_destination – множитель приемника. Задает степень влияния альфа-компоненты на цвет приемника

Исходный фрагмент — отрисовываем сейчас, а целевой фрагмент — уже находится во фреймбуфере

Настройка коэффициентов смешивания

gl.BlendFunc(GLenum sfactor, GLenum dfactor)

Первый параметр функции **gl.blendFunc** определяет исходный множитель, а второй — целевой множитель.

Параметр	Значение множителя
GL_ZERO	0
GL_ONE	1
GL_SRC_COLOR	$\overline{C}_{source}$
GL_ONE_MINUS_SRC_COLOR	$1 - \overline{C}_{source}$
GL_DST_COLOR	$\overline{C}_{destination}$
GL_ONE_MINUS_DST_COLOR	$1 - \overline{C}_{destination}$
GL_SRC_ALPHA	Равен альфа-компоненте *alpha* вектора $\overline{C}_{source}$
GL_ONE_MINUS_SRC_ALPHA	1 — *alpha* вектора $\overline{C}_{source}$
GL_DST_ALPHA	Равен альфа-компоненте *alpha* вектора $\overline{C}_{destination}$
GL_ONE_MINUS_DST_ALPHA.	1 — *alpha* вектора $\overline{C}_{destination}$
GL_CONSTANT_COLOR	Равен вектору цвета $\overline{C}_{constant}$
GL_ONE_MINUS_CONSTANT_COLOR	$1 - \overline{C}_{constant}$
GL_CONSTANT_ALPHA	Равен альфа-компоненте *alpha* вектора $\overline{C}_{constant}$
GL_ONE_MINUS_CONSTANT_ALPHA	1 — *alpha* вектора $\overline{C}_{constant}$

Пример 1

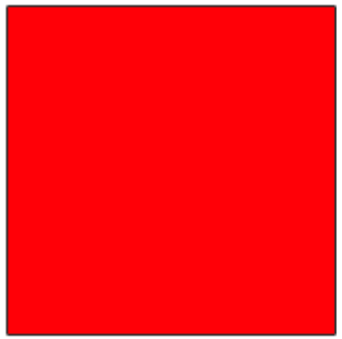
Значение прозрачности целевого фрагмента постоянное значение — единица (непрозрачный)
Исходный фрагмент — отрисовываем сейчас, а целевой фрагмент — уже находится во фреймбуфере

Если **gl.blendFunc**(gl.SRC_ALPHA, gl.ONE);

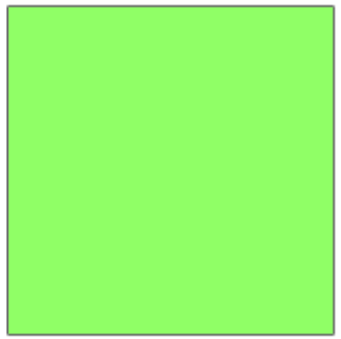
То будет так

$R_{result} = R_s * A_s + R_d$
$G_{result} = G_s * A_s + G_d$
$B_{result} = B_s * A_s + B_d$
$A_{result} = A_s * A_s + A_d$

Пример 2. Как получить такое смешивание?

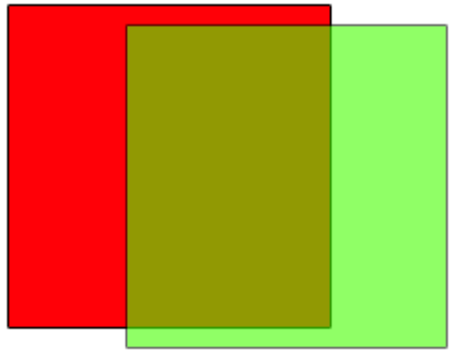


(1.0, 0.0, 0.0, 1.0)



(0.0, 1.0, 0.0, 0.6)

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$



(0.4, 0.6, 0.0, 0.76)

Следует выбрать такие параметры, чтобы коэффициент источника равнялся alpha (значение альфа-компоненты) цвета источника, а коэффициент приемника равнялся 1 – alpha.
Что равнозначно вызову:
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

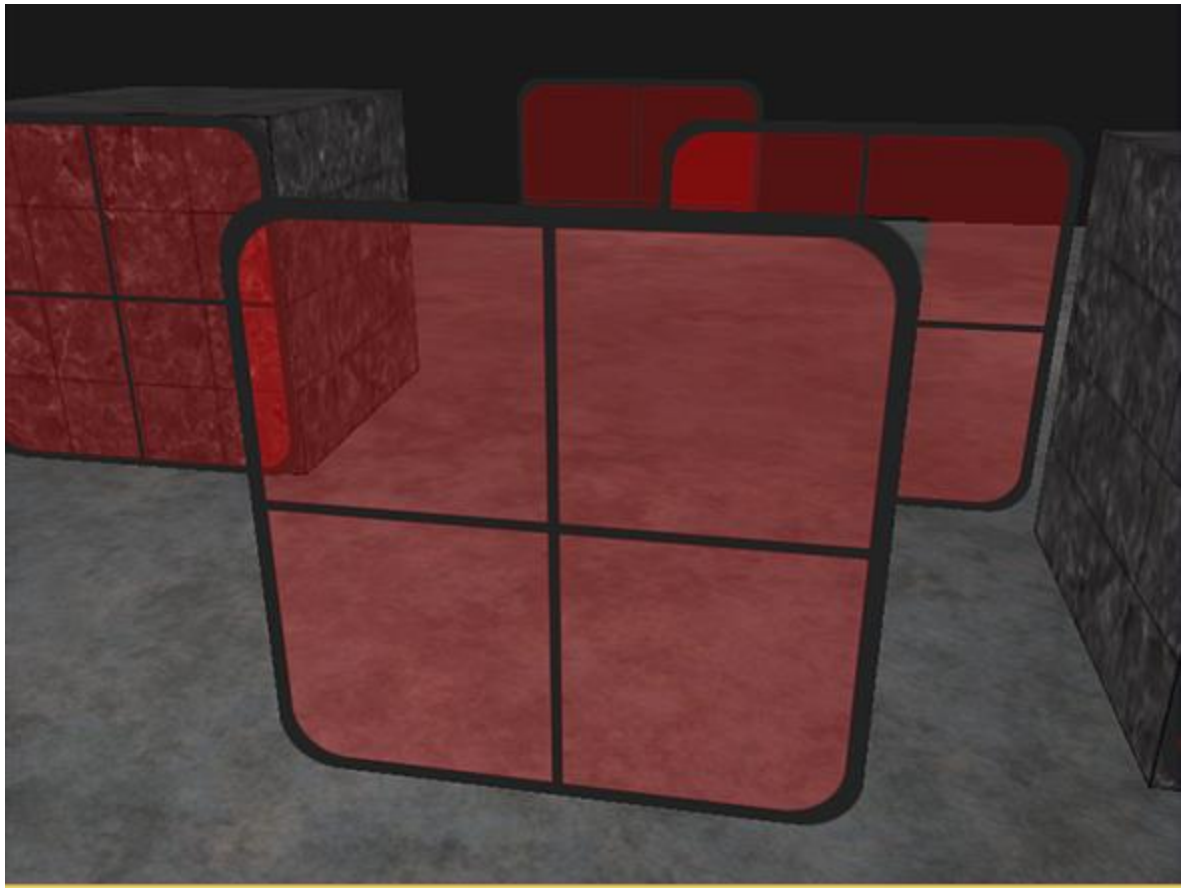
Фрагментный шейдер с текстурой и прозрачностью

gl.enable(gl.BLEND); //смешивание выключено по умолчанию

gl.disable(gl.DEPTH_TEST); // отключить проверку глубины
Предупреждение: Отключать depth test для прозрачных объектов нужно осторожно, обычно сортируют объекты, а depth test оставляют включенным для непрозрачных частей.

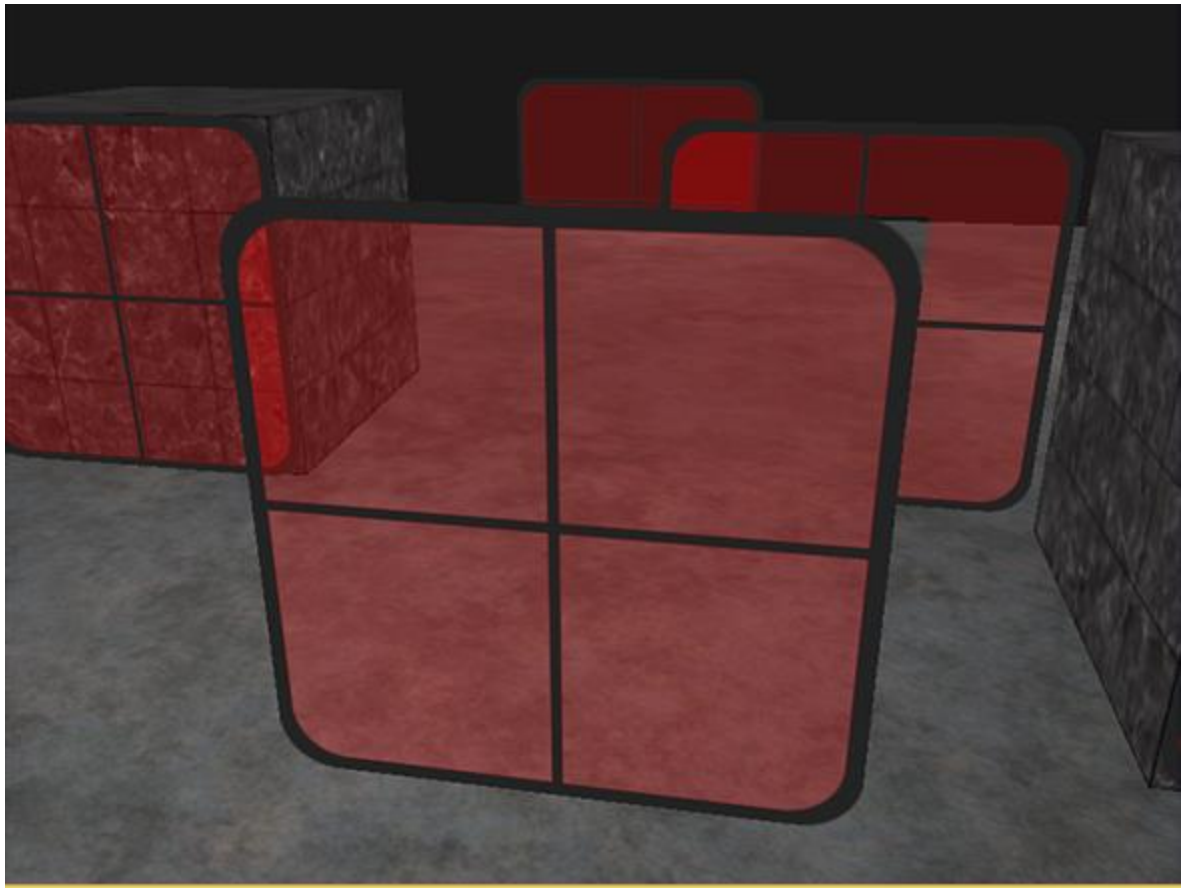
```
in vec2 vTextureCoord;
in vec3 vLightWeighting;

uniform float uAlpha;

uniform sampler2D uSampler;

void main(void) {
    vec4 textureColor = texture(uSampler, vec2(vTextureCoord.s, vTextureCoord.t));
    gl_FragColor = vec4(textureColor.rgb * vLightWeighting, textureColor.a * uAlpha);
}
```

Артефакты. Вопрос: почему так получается?



Устанавливаем исходный множитель в SRC_ALPHA, а целевой множитель в ONE_MINUS_SRC_ALPHA.

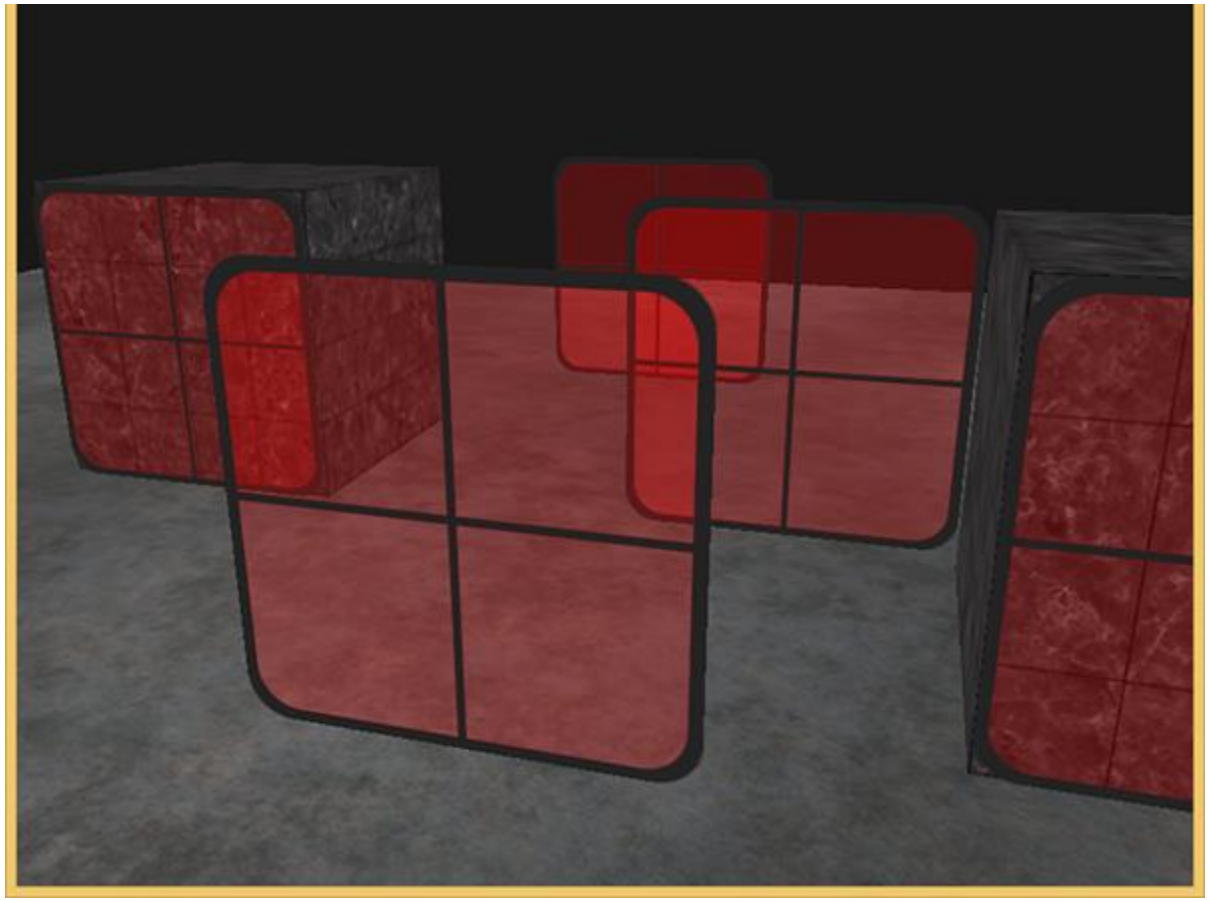
Артефакты. Ответ на вопрос предыдущего слайда



Устанавливаем исходный множитель в SRC_ALPHA, а целевой множитель в ONE_MINUS_SRC_ALPHA.

Но исходный и целевой фрагмент обрабатываются по-разному, и поэтому остается зависимость от порядка отрисовки.

Важно: порядок отрисовки имеет значение!



Так как же получить «настоящую» прозрачность?

Непрозрачные объекты — рисуем первыми (с включенным depth test)

Прозрачные и полупрозрачные объекты — рисуем последними, отсортированными от дальних к ближним

Иначе прозрачные объекты за ближайшими будут не видны или смешаются неправильно

Индустриальный пайплайн в кино и играх

Основные этапы создания и визуализации 3D моделей в кино и game-индустриях

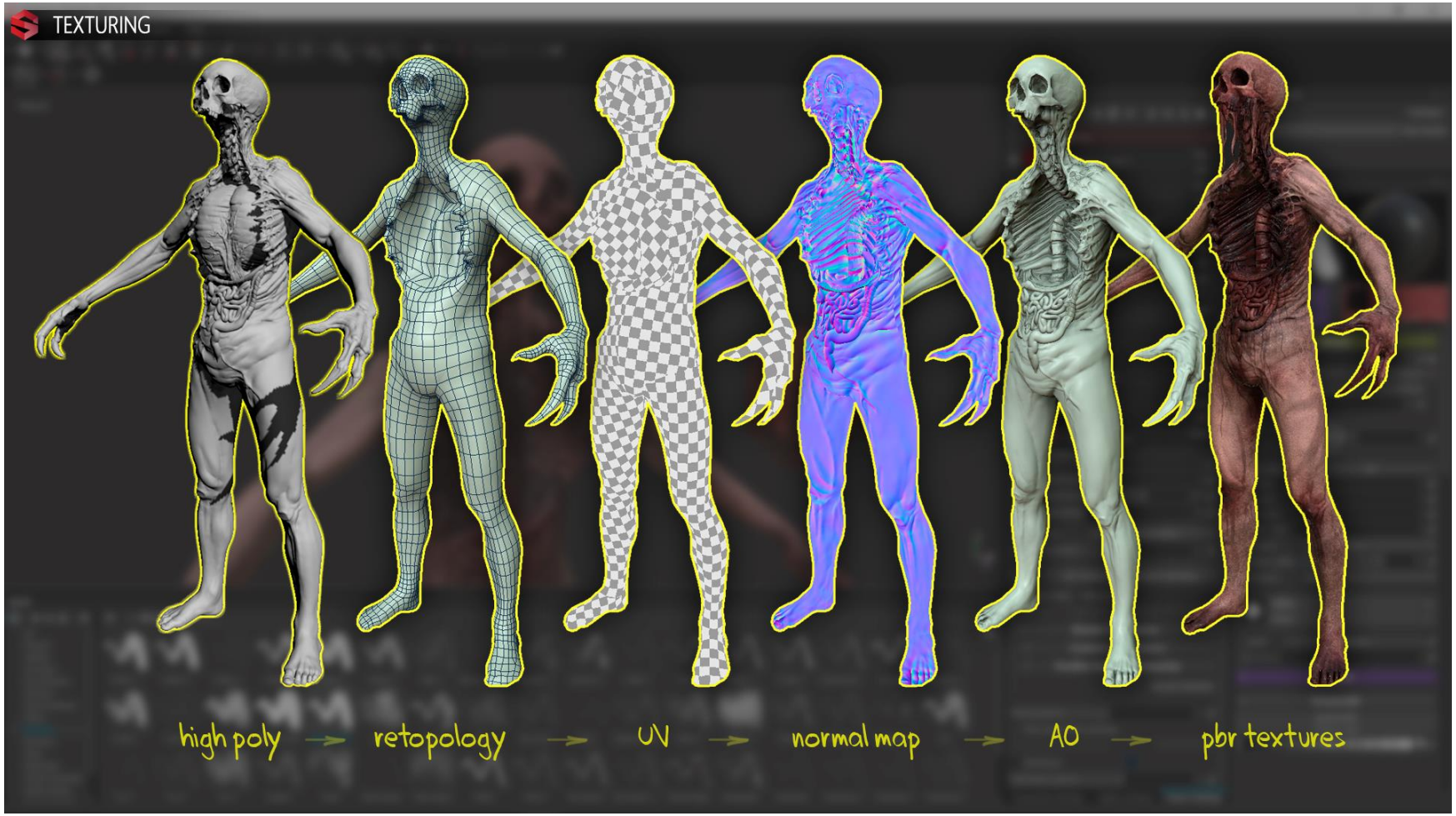
- Моделирование – создание трехмерных объектов
- Текстурирование – наложение текстур и материалов на 3D-модели
- Риггинг – создание виртуального скелета, набора костей/суставов для последующей анимации персонажа
- Анимация – оживление, анимирование трёхмерного персонажа
- Рендеринг – 3D визуализация созданной графики
- Композитинг – объединение отдельных элементов в финальную сцену. К примеру, интегрирование 3D сцен в съёмочный материал, цветокоррекция и добавление эффектов

Моделирование: кино vs игры

В фильмах	В играх
можно использовать криволинейные поверхности (NURBS-моделирование) и полигоны (полигональное моделирование)	обычно используют только полигональные модели, их проще всего визуализировать
обычно создают высокополигональные модели, рендеринг которых, проходит по несколько часов, а то и дней	используются низкополигональные модели, визуализация происходит прямо по ходу игры. Часто в компьютерных играх встречается LOD-технология (Level of Detail – уровень детализации)

3D-редакторы Autodesk Maya, Autodesk 3Ds Max, Cinema 4D, Modo и Blender (бесплатный)
для цифрового скульптинга: ZBrush, Mudbox, 3D Coat, Sculptris

Текстурирование. Виды текстур



Создаётся набор текстур:

- цвет
- карта неровностей (bump)
- карта нормалей (normal map – создаёт видимость рельефа)
- карта рельефа (displacement – создаёт реальный рельеф)
- карта бликов (specular)
- карта прозрачности (alpha)
- и т д.

Зачем нужна ретопология в 3D моделировании?

Оптимизация производительности (уменьшение числа полигонов)
Высокополигональные модели могут быть слишком тяжелыми для рендеринга в реальном времени, особенно в играх

Упрощение анимации (правильная сетка деформируется лучше)
Модели с правильной топологией легче анимировать. Хорошо организованная сетка позволяет избежать проблем с деформацией при движении

Текстурирование и UV-развертка
Правильная топология облегчает процесс создания UV-разверток и текстурирования модели. Это особенно важно для создания реалистичных текстур, которые точно соответствуют форме модели

Сохранение деталей при уменьшении сложности
Ретопология позволяет сохранить важные детали модели, при этом уменьшая количество полигонов

PBR (Physically Based Rendering)

Метод рендеринга, основывающийся на физических свойствах материалов и их взаимодействии со светом

Основные карты PBR-текстур (материалов):

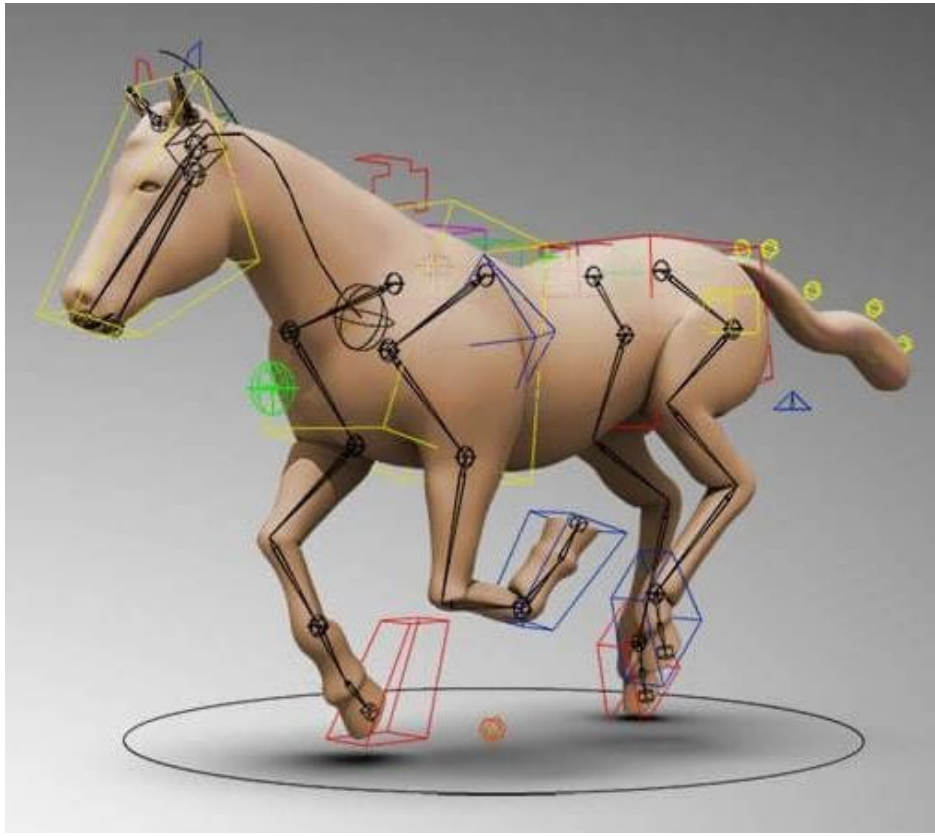
Albedo (или Diffuse) Основной цвет материала при нейтральном освещении
Metallic Карта показывает, является ли материал металлическим
Roughness Карта шероховатостей
Normal Map Эта карта добавляет детали на поверхность, создавая иллюзию неровностей и различных текстур. Не изменяет геометрию объекта, но позволяет добавлять мелкие детали. Например, царапины, небольшие углубления и т. д., что делает поверхность более реалистичной
Ambient Occlusion — затенение в углах

PBR-текстуры используются в различных проектах, от игр до архитектурной визуализации

Риггинг

Риггинг – создание «скелета», костей модели

Занимаются этим в кино и game-индустрии художники по оснастке модели, **сетаперы** (Setup artist). Еще их называют **skinning, rigging artist**



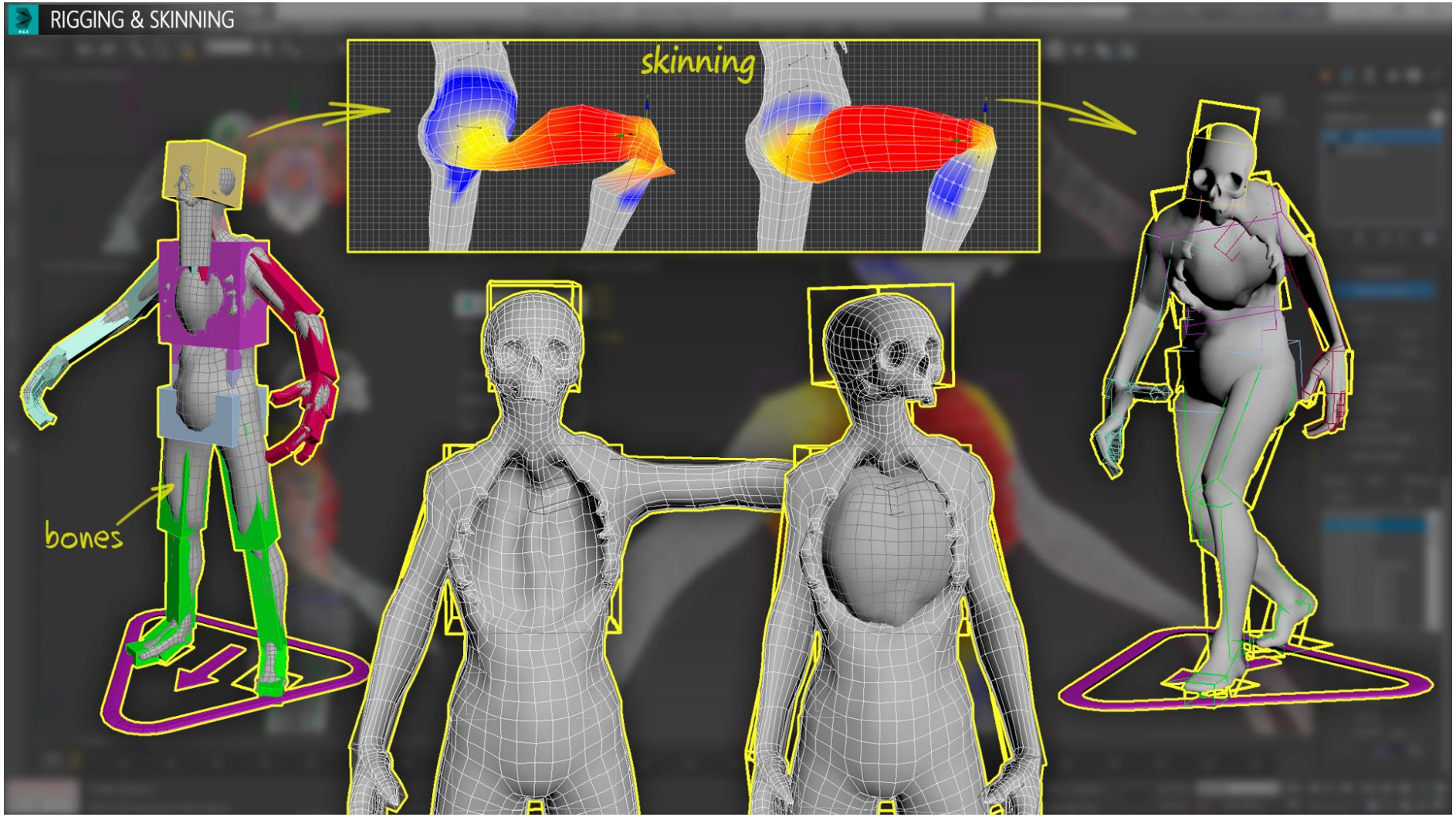
Сетаперы создают кости и контроллеры для управления этими костями, с помощью которых аниматоры могут оживить модель

В **кино** обычно создаётся множество сложных контроллеров для аниматоров. Например, для **лицевой анимации** (facial control rig) и мимики модели

В **играх** можно **обойтись** и без них, если персонаж не разговаривает в игре

Для риггинга модели подойдут те же 3D-редакторы, большинство из которых – комплексные 3D-пакеты.

Риггинг. Эффекты и деффекты



Анимация

Главная задача аниматора – сделать движения модели максимально реалистичными

Анимация по ключевым кадрам (Keyframes): положение в промежуточных кадрах вычисляется программой

Процедурная анимация: используется специальная программа для управления персонажем

Технология **Motion Capture** (система захвата движений): подразумевает наложение движений реальных актеров на трёхмерных персонажей.

Негласным лидером в создании трёхмерной анимации является Autodesk Maya
Помимо Maya отличные инструменты для анимации – 3Ds Max и Cinema 4D

Форматы 3D-моделей

- OBJ + MTL (материалы для OBJ)
  - очень простой
  - является текстовым
  - не поддерживает анимацию
- COLLADA
  - COLLAborative Design Activity
  - основан на XML
  - поддерживает анимацию
- 3DS, MAX и BLEND
- бесчисленное количество форматов, используемых в играх и других приложениях
  - Quake I (.mdl), Quake II (.md2), Quake III Mesh (.md3), Quake III Map/BSP (.pk3)
  - Doom 3 (.md5*), Unreal (.3d)
  - PovRAY Raw (.raw), Terragen Terrain (.ter), 3D GameStudio/3DGS (.mdl)

Структура OBJ-файла

```
# Вершины (координаты)
v 0.123 0.234 0.345
v 0.456 0.567 0.678

# Текстурные координаты
vt 0.500 0.500
vt 0.750 0.250

# Нормали
vn 0.707 0.000 0.707
vn 0.000 1.000 0.000

# Полигоны (формат: вершина/текстура/нормаль)
f 1/1/1 2/2/2 3/3/3
```

MTL — материалы

```
# MTL (Material Library)

newmtl MetalMaterial
    Ka 0.2 0.2 0.2    # ambient color
    Kd 0.8 0.8 0.8    # diffuse color
    Ks 1.0 1.0 1.0    # specular color
    Ns 50.0          # shininess
    map_Kd metal.png  # diffuse texture
    map_Bump metal_norm.png
```

Формат COLLADA

COLLADA (.dae)

- XML-формат, стандарт ISO
- Поддерживает геометрию, материалы, анимацию, физику
- Используется для обмена между редакторами
- Поддержка в Unreal, Unity, Blender, Maya

Загрузка моделей в WebGL

Open Asset Import Library (Assimp)

- Кроссплатформенная библиотека на C++
- Поддерживает >50 форматов (OBJ, COLLADA, 3DS, FBX)
- Импортирует геометрию, материалы, анимацию
- Используется в коммерческих и open-source проектах

https://assimp.org/

Three.js — загрузка OBJ

```
import { OBJLoader } from 'three/addons/loaders/OBJLoader.js';

const objLoader = new OBJLoader();
objLoader.load('model.obj', (root) => {
   scene.add(root);
});



// Что делает Three.js внутри?
// 1. Парсит OBJ-файл
// 2. Создаёт буферы (позиции, нормали, UV)
// 3. Компилирует шейдеры
// 4. Всё то, что мы делали вручную в Лекциях 2-3!
```

Three.js — загрузка OBJ + MTL

```
import { OBJLoader } from 'three/addons/loaders/OBJLoader.js';
import { MTLLoader } from 'three/addons/loaders/MTLLoader.js';

const mtlLoader = new MTLLoader();
mtlLoader.load('model.mtl', (mtl) => {
    mtl.preload();
    const objLoader = new OBJLoader();
    objLoader.setMaterials(mtl);
    objLoader.load('model.obj', (root) => {
        scene.add(root);
    });
});
```

Путь модели: от файла до экрана

[OBJ/MTL файл]
↓ (Assimp / Three.js)
[JavaScript объекты (массивы вершин, нормалей, UV)]
↓ (gl.bufferData, из Лекции 2)
[VBO в видеопамяти]
↓ (gl.vertexAttribPointer, из Лекции 2)
[Вершинный шейдер (позиция + нормаль + UV)]
↓ (интерполяция)
[Фрагментный шейдер]
↓ + [Текстура] (из Лекции 4) + [Освещение] (из Лекции 3)
[Пиксель на экране]

Оптимизация данных (VBO, VAO, IBO)
Опять про буферы

VBO — Vertex Buffer Object

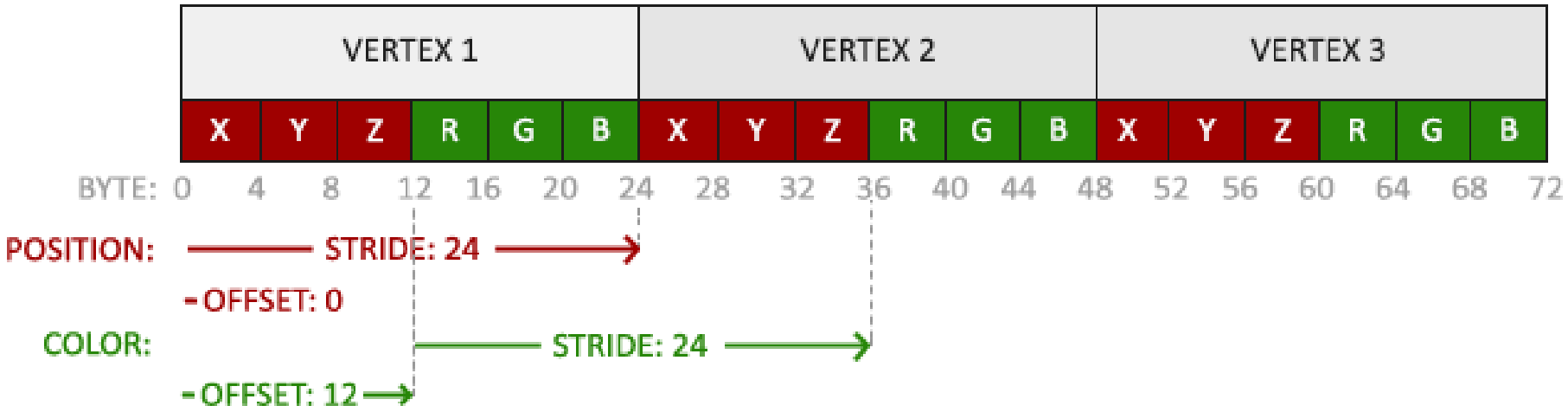
- Буфер в видеопамяти GPU
- Хранит атрибуты вершин (позиции, нормали, UV, цвета)
- Позволяет не передавать данные из CPU каждый кадр
- Создаётся через gl.createBuffer() и gl.bufferData()

- Можно создать много VBO, и каждый VBO имеет свой уникальный идентификационный идентификатор в OpenGL.
- Этот идентификатор соответствует конкретному адресу видеопамяти VBO.
- Через этот идентификатор можно получить доступ к данным в конкретном VBO.

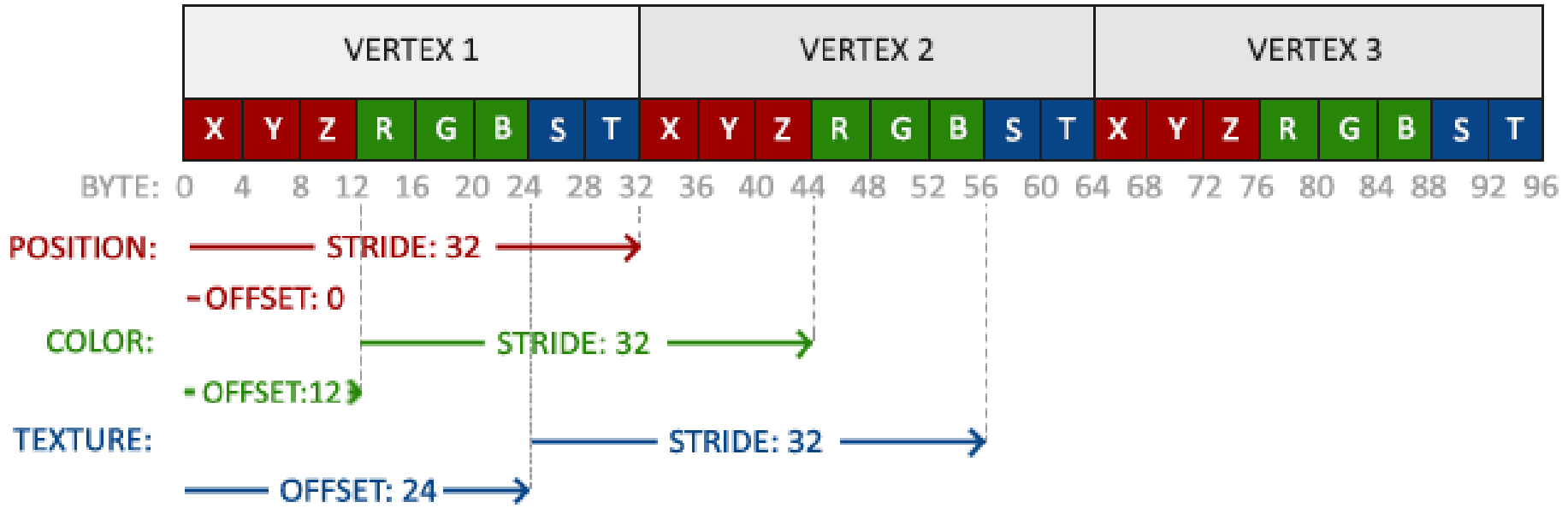
Формат вершинного буфера

Плотно упакованный буфер

VBO в памяти: координаты и цвет



VBO в памяти: координаты, цвет и текстура



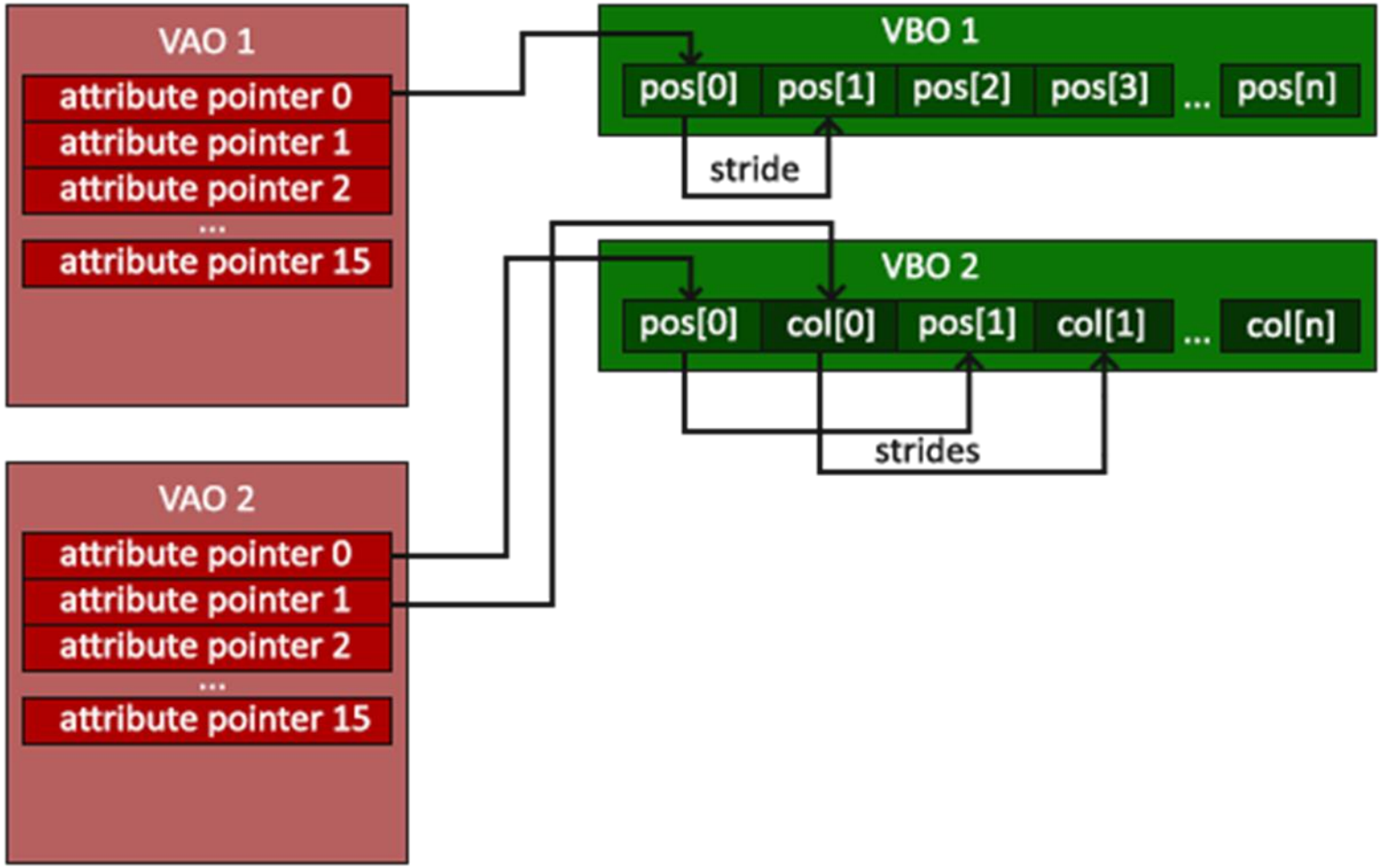
VBO: координаты, цвет и текстура

```
gl.EnableVertexAttribArray(0);
gl.EnableVertexAttribArray(1);
gl.EnableVertexAttribArray(2);

// Подключаем VBO
gl.BindBuffer(GL_ARRAY_BUFFER, VBO);

// Атрибут с координатами
gl.VertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)0);
// Атрибут с цветом
gl.VertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(3* sizeof(GLfloat)));
// Атрибут с текстурой
gl.VertexAttribPointer(2, 2, GL_FLOAT,GL_FALSE,   8 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
```

VAO (объект Vertex Array)

• Хранит настройки атрибутов:
  - Какие атрибуты включены (gl.enableVertexAttribArray)
  - Как они расположены (gl.vertexAttribPointer)
  - Формат данных вершин
  - Какой VBO с каким атрибутом связан

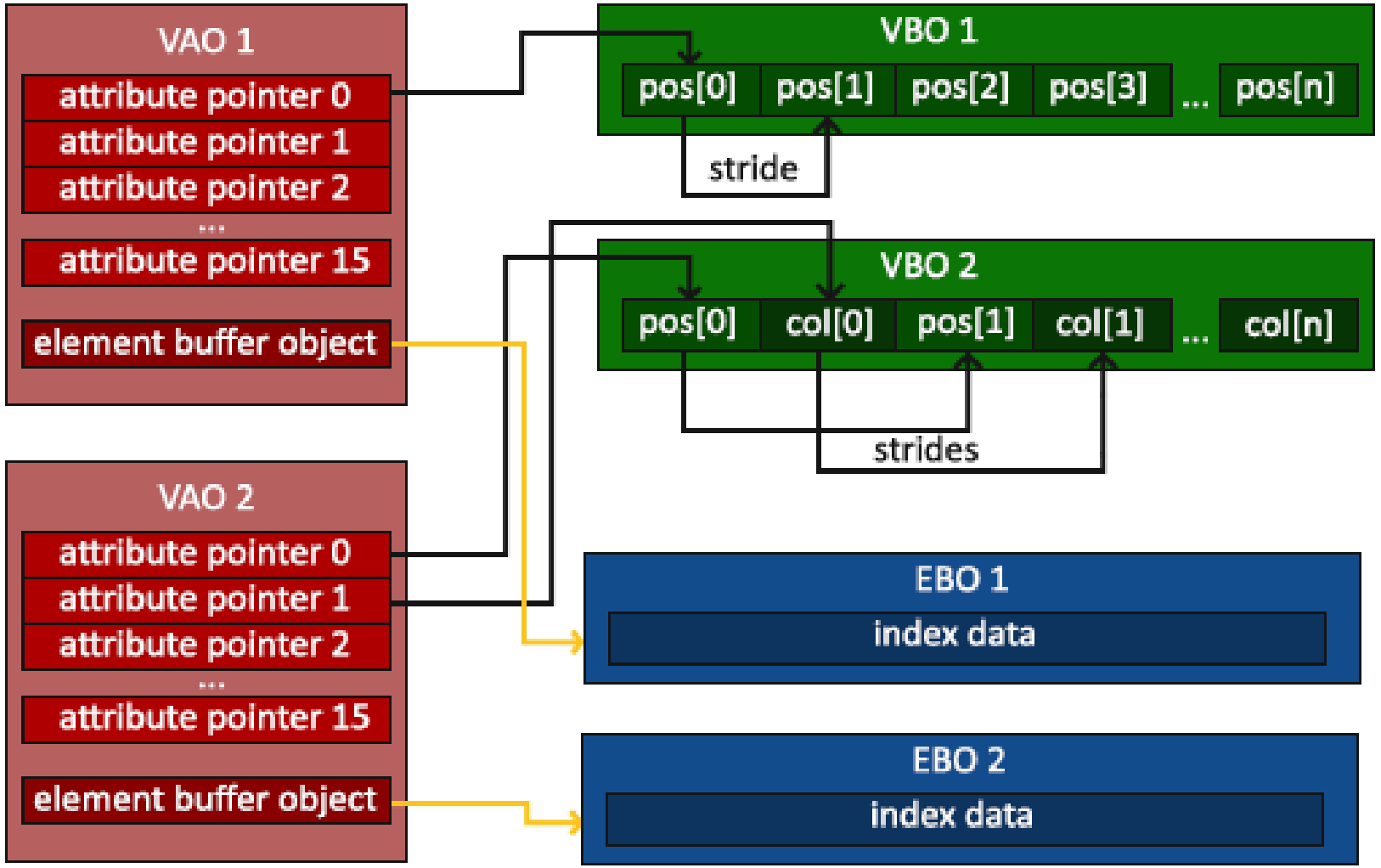
• Позволяет переключаться между конфигурациями одной командой
• Упрощает код и повышает производительность



IBO/EBO — Index Buffer Object

- Хранит индексы вершин — Аналог индексного массива для массива вершин
- Позволяет повторно использовать вершины
- Экономия памяти (24 вершины с индексами вместо 36 без)
- Используется с gl.drawElements()

VAO с VBO и IBO (EBO)



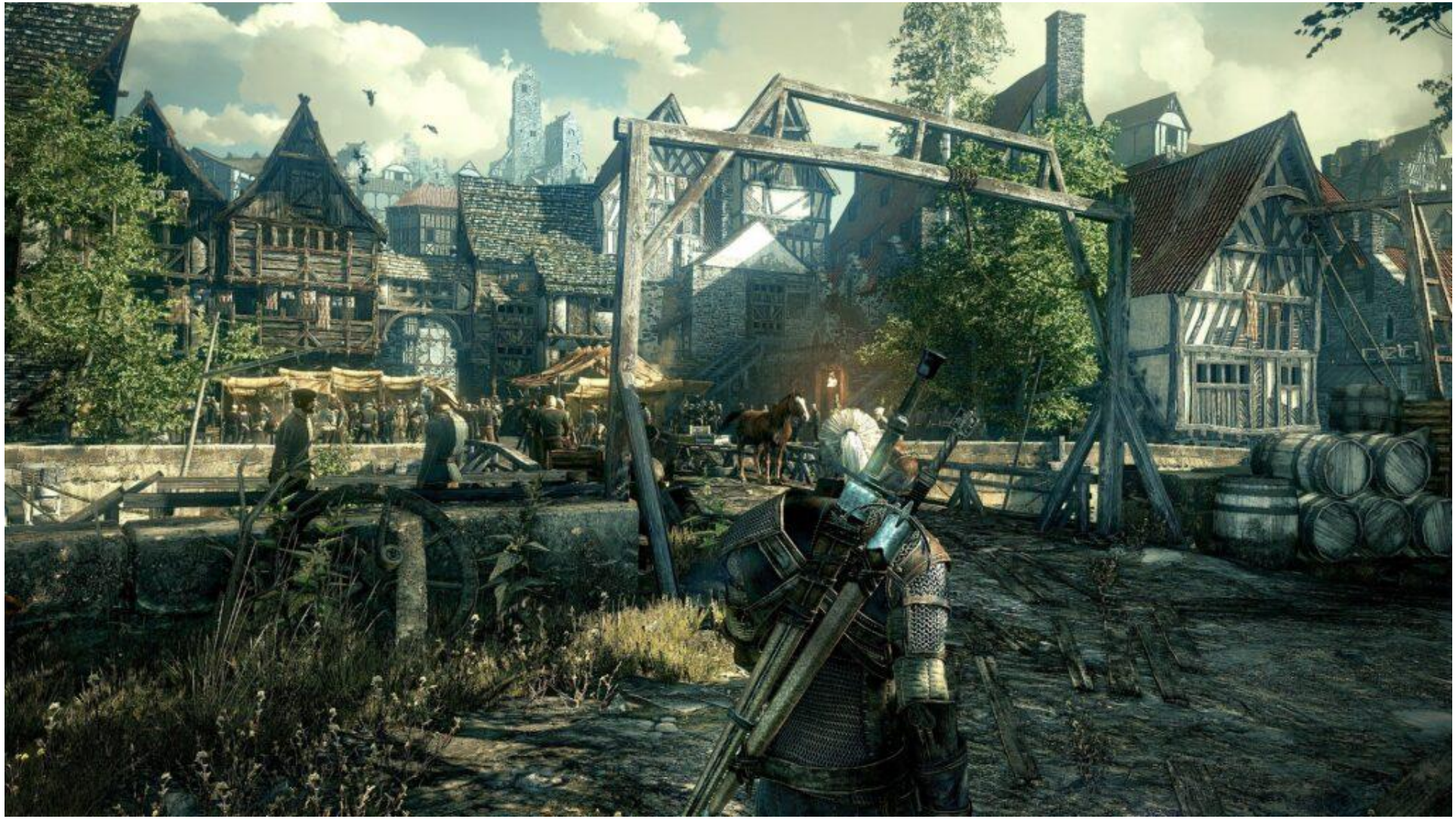
```
// ..:: Код инициализации :: ..
   // 1. Привязываем VAO
   gl.BindVertexArray(VAO);
   // 2. Копируем наши вершины в буфер для WebGL
   gl.BindBuffer(GL_ARRAY_BUFFER, VBO);
   gl.BufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
   // 3. Копируем наши индексы в в буфер для OpenGL
   gl.BindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);
   gl.BufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
   // 3. Устанавливаем указатели на вершинные атрибуты
   gl.VertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
   gl.EnableVertexAttribArray(0);
   // 4. Отвязываем VAO
   gl.BindVertexArray(0);

[...]

// ..:: Код отрисовки (в игровом цикле) :: ..
gl.UseProgram(shaderProgram);
gl.BindVertexArray(VAO);
gl.DrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
gl.BindVertexArray(0);
```

К чему мы стремимся? Примеры игровых миров

- Ведьмак 3: Дикая охота
- Death Stranding
- Локации в играх жанра hidden object
- и др.

Все эти миры построены из моделей, текстур и освещения —
тех самых компонентов, которые мы изучаем в курсе!

Ведьмак 3: Дикая охота



Death Stranding

Локации в играх жанра hidden object

Что мы сегодня разобрали

Теперь мы умеем загружать готовые модели и накладывать на них текстуры.

Следующий шаг: добавление реализма без усложнения геометрии.

1. Текстуры в WebGL
   - Загрузка изображений, CORS, параметры фильтрации
   - MIP-карты и степени двойки
   - UV-координаты и их интерполяция

2. Прозрачность и смешивание
   - discard для отбрасывания фрагментов
   - Формула смешивания и gl.blendFunc
   - Важность порядка отрисовки

3. Индустриальный пайплайн
   - Этапы создания 3D-контента
   - PBR-текстуры (Albedo, Normal, Roughness...)

4. Форматы и загрузка моделей
   - Структура OBJ и MTL
   - COLLADA как универсальный формат
   - Загрузка через Three.js и Assimp

5. Оптимизация данных
   - VBO, VAO, IBO — как данные хранятся в GPU

Вопросы для самопроверки

- Чем отличается LINEAR от NEAREST? Когда что использовать?
- Зачем нужны MIP-карты?
- Почему размеры текстур часто делают степенями двойки?
- Что произойдет, если не включить gl.blendFunc для прозрачных объектов?
- Как связаны UV-координаты из OBJ-файла и атрибут aTextureCoord в шейдере?
- Зачем нужны VAO, если есть VBO?

- (Для продвинутых) Как Three.js внутри превращает OBJ-файл в буферы WebGL?