Упражнение 2 Базов проект с LWJGL

Lightweight Java Game Library

LWJGL (Lightweight Java Game Library) е популярна библиотека с отворен код за създаване на игри и мултимедийни приложения на Java. Тя предоставя достъп до мощни графични, аудио и входно-изходни библиотеки, като използва ниско ниво на взаимодействие с хардуера чрез OpenGL, Vulkan, OpenAL и други. LWJGL е често използвана от разработчиците на 3D игри и графични приложения, тъй като улеснява достъпа до функциите на GPU (графичния процесор).

LWJGL (Lightweight Java Game Library) е библиотека, която осигурява директен достъп до API-та на ниско ниво като OpenGL, OpenAL и GLFW в Java. Това предоставя възможност да се научат основните принципи на рендериране, графични трансформации, осветеност и оптимизация.

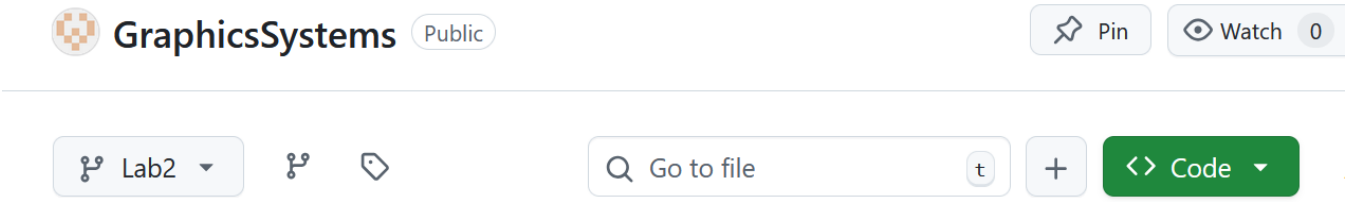
LWJGL позволява директна работа с OpenGL, което дава дълбоко разбиране за това как работи графичният хардуер. В сравнение с библиотеки като JavaFX или Unity, LWJGL изисква ръчно управление на буфери, шейдъри и рендериране, което е полезно за фундаменталното разбиране на 3D графиката.

Библиотеката е налична на следния адрес: https://www.lwjgl.org/customize. За упражненията по дисциплината „Графични системи“ са необходими следните нейни компоненти:

- **Библиотека GLFW** – осигурява създаването на графични прозорци и управлението на входни устройства при приложения с OpenGL;
- **Библиотека OpenGL** – осигурява достъп до хардуерно ускорено рендиране чрез видеокартата, което позволява бързо и ефективно визуализиране на 2D и 3D обекти. То предоставя механизми за управление на графичния конвейер – върхове, буфери, шейдъри, текстури и трансформации – които са основата за изграждане на интерактивни графични приложения и игрови енджини;
- **Библиотека stb** – осигурява лесен и ефективен начин за обработка на текстури;
- **Библиотека JOML** – библиотека за линейна алгебра, предназначена за 2D и 3D графични приложения в Java. Тя предоставя класове за вектори, матрици, кватерниони и трансформации, които се използват за изчисляване на позиция, ротация, мащабиране и проекции в графичния конвейер.

Базови проекти

По време на семестъра ще работим на базата на предварително подготвени проекти, които са конфигурирани с помощта на Maven, с включени базови класове за изпълнение на възложените задания. Всички необходими библиотеки ще бъдат включени в конфигурационния *pom* файл на проекта. Вашата задача ще бъде да допълните базовите проекти по начин, по който да се визуализират и упражнят изучаваните в съответното занятие концепции и алгоритми. Проектите се публикуват в хранилище https://github.com/donikast/GraphicsSystems, като базата на всяко упражнение е публикувана в собствен branch.



За работа в часовете в зала 301 ТВ използваме Windows профила **Java**. В директория **GraphicsSystems** създайте **папка с Вашият факултетен номер** и **помествайте в нея проектите**, по които работите.

Базов проект за упражнение 2

Базовият проект, с който ще работим по време на упражнение 2, има за цел да запознае студентите с основната структура на проектите през семестъра и логиката, която е имплементирана в тях. Може да бъде изтеглен от следния адрес: https://github.com/donikast/GraphicsSystems/tree/Lab2 .

Основните моменти, заложени в проекта, са както следва:

1) Управление на графичния прозорец и свързани с него събития
2) Реализация на двигател (Engine) с основен рендериращ цикъл
3) Абстрактен рендериращ интерфейс и OpenGL имплементация
4) Геометрия и GPU буфери (Mesh слой)
5) Моделен слой за композиция на графични елементи
6) Организация и управление на сцена
7) Шейдъри и шейдърна програма

Управление на графичния прозорец и свързани с него събития

Графичният прозорец е интерфейсът между графичното приложение и операционната система. Изпълнява няколко основни функции:

- Осигурява повърхност за визуализация (екранно пространство).
- Създава и поддържа графичен контекст.
- Приема и обработва потребителски събития (затваряне, клавиатура, мишка).
- Управлява обмена на кадри между CPU и GPU.

В настоящия проект се реализира посредством класа **Window**.

За реализацията на свързаните с графичния прозорец и OpenGL контекста функционалности в употреба влиза библиотеката GLFW. Програмната логика на класа включва няколко основни момента:

1. Създаване на графичния прозорец. Включва:
    - Инициализация на библиотеката (с помощта на метода `initGLFW()`)

```
private void initGLFW() {
    if (!glfwInit()) {
        throw new IllegalStateException("GLFW initialization
failed");
```

```
    }
}
```

- Създаване на графичен прозорец с подадените параметри. Връща идентификатор (handle), който се използва при всички последващи операции. Ако създаването не успее, се хвърля изключение.

```
private long createWindow() {
    long windowHandle = glfwCreateWindow(width, height, title,
0, 0);
    if (windowHandle == 0) {
        throw new RuntimeException("Window creation failed");
    }
    return windowHandle;
}
```

- Инициализация на OpenGL контекст

Методът initContext() прави прозореца текущ OpenGL контекст и създава OpenGL capabilities чрез createCapabilities(). Последното позволява използването на OpenGL функции в приложението.

```
private void initContext() {
    glfwMakeContextCurrent(handle);
    createCapabilities();
}
```

- Конфигурация на прозореца

В текущата реализация се активира вертикална синхронизация (V-Sync). Това синхронизира честотата на кадрите с честотата на монитора и предотвратява визуални артефакти като screen tearing (на екрана се виждат „разкъсани“ хоризонтални линии, защото различни части от изображението принадлежат на различни кадри).

```
private void configureWindow() {
    glfwSwapInterval(1);
}
```

2. Управление по време на изпълнение

Методът update() изпълнява две ключови операции:

- glfwSwapBuffers(handle) – разменя front и back buffer (двойно буфериране). OpenGL използва двойно буфериране: front buffer – показва текущия кадър и back buffer – в него се рисува следващият кадър. След приключване на рендирането буферите се разменят. Това предотвратява трептене и осигурява плавна анимация.
- glfwPollEvents() – обработва всички чакащи събития от операционната система. Проверява за: затваряне на прозореца, вход от клавиатура, движения на мишката и други системни събития.

```
public void update() {
    glfwSwapBuffers(handle);
    glfwPollEvents();
}
```

Този метод се извиква при всеки цикъл на енджина и е съществена част от рендериращия процес.

3. Освобождаване на ресурси

Методът destroy() унищожава прозореца и прекратява работата на GLFW.

Моделен слой за композиция на графични елементи

Моделният слой в проекта служи за логическо групиране и композиция на графични елементи в по-сложни обекти. Той позволява комбиниране на множество геометрични елементи в един логически обект. Основната му цел е да осигури структурирано представяне на сложни графични обекти чрез композиция на по-прости елементи. Неговата реализация включва класовете ModelElement и Model.

Организация и управление на сцена

Сцената представлява логически контейнер, който организира всички графични обекти, подлежащи на визуализация. Тя изпълнява ролята на централен регистър на обектите в приложението и осигурява структурирано управление на визуалното съдържание. Основната ѝ цел е да групира, управлява и предоставя достъп до всички графични обекти, които трябва да бъдат рендирани. В проекта нейната реализация включва класовете Scene, SceneObject (и наследници) и SceneBuilder.

Геометрия и GPU буфери (Mesh слой)

Основната цел на този слой е да преобразува масив от данни за върховете в GPU-структура. За изясняване на слоя предварително е необходимо студентите да са наясно с два основни термина: Vertex Buffer Object (VBO) и Vertex Array Object (VAO). VBO е буфер в графичната памет, който съхранява всички върхове на обекта. Това включва: координати (x, y, z), цветове (r, g, b, a), текстурни координати (u, v), нормали за осветление (nx, ny, nz). OpenGL не може директно да работи с float[] масиви в Java. VBO изпраща върховете в графичната памет, за да може GPU да ги обработва бързо. Това позволява ефективен рендеринг без излишни копирания от RAM към GPU.

VAO е контейнер, който съхранява всички настройки на върховете, включително: какви атрибути има всеки връх (позиция, текстура, нормали); къде в паметта се намират (кои VBO-та се използват); как OpenGL да ги интерпретира.

Mesh слоят е разделен на няколко компонента с ясно разграничени роли.

1) Клас **Geometry** – описва формата и организацията на върховите данни, които впоследствие се използват от Mesh слоя за конфигуриране на VAO (Vertex Array Object). Класът описва как са организирани данните, какъв е форматът на върховете и как трябва да бъдат интерпретирани от GPU.

2) Клас **VertexBuffer** – капсулира създаването и управлението на VBO (Vertex Buffer Object). Неговата роля е да създаде буфер в GPU паметта, да качи данните на върховете и да осигури bind/unbind операции.
3) Класът **VertexAttribute** описва локацията на дадения атрибут в шейдъра и броя на компонентите в този атрибут. Това позволява гъвкаво дефиниране на различни типове върхове, например:

   - само позиция,
   - позиция + цвят,
   - позиция + нормала и т.н.

4) **DrawMode** – позволява избор на примитив за изчертаване.
5) Клас **Mesh** - реализира реалното свързване между Geometry и GPU. При създаване той:
   1. Генерира VAO (Vertex Array Object).
   2. Създава и bind-ва VBO.
   3. Конфигурира vertex атрибутите чрез glVertexAttribPointer.
   4. Активира атрибутите.
   5. Запомня броя върхове.
   6. Освобождава VAO.

Така Mesh подготвя цялата конфигурация, необходима за изчертаване на геометрията.

При извикване на mesh.render(); се изпълнява:

```
glBindVertexArray(vao);
glDrawArrays(...);
```

Това задейства графичния pipeline.

Шейдъри и шейдърна програма

Шейдърите са малки програми, които работят директно на графичния процесор (GPU) и определят как се обработват и изобразяват пикселите и върховете в OpenGL. Те заменят стария OpenGL "Fixed Pipeline" (glBegin/glEnd) и позволяват по-голяма гъвкавост в рендирането.

OpenGL използва няколко типа шейдъри, но основните са:

- Vertex Shader - обработва всеки връх (vertex) на геометрията; определя позицията на върха в пространството; може да предава данни като нормали, UV координати, цветове към следващите етапи.
- Fragment Shader - обработва всеки пиксел от примитивите (триъгълници, линии, точки). Определя цвета на всеки пиксел. Позволява прилагане на текстури, осветление и ефекти.

В проекта създайте директория res/shaders и в тях поместете следните файлове:

vertexShader.vert

```
#version 460 core
layout (location = 0) in vec3 aPos;

void main() {
   gl_Position = vec4(aPos, 1.0);
}
```

fragmentShader.frag

```
#version 460 core
out vec4 FragColor;

void main() {
   FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Шейдърите в OpenGL са написани на GLSL (OpenGL Shading Language). GLSL (OpenGL Shading Language) е C-подобен език, предназначен за програмиране на GPU. Работи директно върху GPU, като обработва върхове, пиксели, текстури, светлина и др.

Взаимодействието с шейдърите се реализира с помощта на класа **ShaderProgram**. Класът ShaderProgram отговаря за създаването, компилирането и свързването на vertex и fragment шейдърите в изпълнима програма, която се използва от GPU по време на рендиране. Той управлява активирането на шейдъра и подаването на uniform променливи, като осигурява връзката между CPU логиката и графичния pipeline.

Абстрактен рендериращ интерфейс (Renderer) и OpenGL имплементация

Renderer слоят поема отговорността за това как сцената се превръща в изображение върху екрана. Този слой получава сцена, обхожда нейните обекти, активира шейдърна програма, подготвя OpenGL състоянието, извиква render върху Mesh и поставя задача на GPU да генерира кадър.

Реализация на двигател (Engine) с основен рендериращ цикъл

Класът **Engine** и неговият наследник **App** представляват централен управляващ компонент на графичната система. Тяхната основна задача е да координират работата между:

- прозореца (Window)
- сцената (Scene)
- рендерера (Renderer / OpenGLRenderer)
- и останалите графични обекти.

Engine реализира т.нар. основен рендериращ цикъл (main loop), който осигурява непрекъснатото обновяване и визуализация на приложението.

<u>Задачи за самостоятелна работа</u>

1. Разгледайте представения код. Опитайте се да рендирате следните изображения: