İSTANBUL ÜNİVERSİTESİ CERRAHPAŞA

BİLGİSAYAR MÜHENDİSLİĞİ

BİLGİSAYAR GRAFİKLERİ 2025 PROJE 1

MUHAMMET TALHA ODABAŞI

İçindekiler

TEMA

Minecraft dünyasında sıkça karşılaşılan 9 adet nesne çizimi yapılmıştır. İlgili düzenlemeler yapılarak Ağac, ev vb. cisimler tanımlanabilir.

SHADER

Shaderlar, grafik işleme biriminde (GPU) çalışarak nesnelerin ekrana nasıl çizileceğini belirler. Bu projede, Phong aydınlatma modeli temel alınarak üç farklı ışıklandırma yöntemi kullanılmıştır: Noktasal ışık (Point Light), Yönlendirilmiş ışık (Directional Light) ve Spot ışık (Spot Light).

Vertex shader, nesnelerin dünya koordinatlarındaki konumlarını hesaplayarak görüntüleme matrisiyle birleştirir ve fragmente aktarılacak normal, renk ve doku koordinatlarını ayarlar. Benim kodumun (**vertex.shader**) temel görevleri:

1. **Veri Girişi:**
    a. **aPos:** Verteksin model koordinatlarındaki konumu.
    b. **aNormal:** Verteksin yüzey normal vektörü (aydınlatma için kullanılır).
    c. **aColor:** Verteksin rengi.
    ç. **aTex:** Verteksin doku (texture) koordinatları.
2. **Dönüşümler:**
    a. **myCurPos:** Model matrisi ile çarpılarak verteksin dünya koordinatlarındaki konumu hesaplanır.
    b. **gl_Position:** Kameranın bakış açısına göre verteksin ekrana nasıl yerleştirileceğini belirler (görüntüleme matrisi ile çarpılır).
3. **Çıkış Değişkenleri:**
    a. **myNormal:** Normaller korunarak fragment shader'a iletilir.
    b. **myFragColor:** Renk verisi fragment shader'a gönderilir.
    c. **myTextureCoord:** Doku koordinatları aktarılır.

Bu shader, her verteksi sahneye uygun şekilde konumlandırarak aydınlatma ve doku işlemleri için gerekli bilgileri fragment shader'a iletir.

Fragment shader, her pikselin rengini hesaplamak için kullanılır ve üç farklı ışıklandırma modeli içerir: Noktasal Işık (Point Light), Yönlendirilmiş Işık (Directional Light) ve Spot Işık (Spot Light). (**fragment.shader**)

1. **Veri Girişi:**
    a. **myFragColor:** Vertex shader'dan gelen renk bilgisi.
    b. **myTextureCoord:** Piksele karşılık gelen doku (texture) koordinatları.
    c. **myNormal:** Yüzeyin normal vektörü (aydınlatma hesaplamaları için).
    ç. **myCurPos:** Pikselin dünya koordinatlarındaki konumu.
    d. **lightPos, camPos:** Işık ve kamera pozisyonları.
    e. **texture0, texture1:** İki farklı doku (texture) bilgisi.
2. **Aydınlatma Modelleri:**
    a. Noktasal Işık (pointLight):
        i. Işığın mesafeye bağlı olarak zayıflamasını hesaplar.

ii. Ortam (ambient), yayılma (diffuse) ve parlama (specular) bileşenlerini içerir.

b. Yönlendirilmiş Işık (directionalLight):

i. Sabit bir ışık yönü kullanır (örn: güneş ışığı gibi).

ii. Mesafeye bağlı zayıflama içermez.

c. Spot Işık (spotLight):

i. Işık belirli bir açı içinde yoğun etki gösterir.

ii. İç ve dış koni açılarına bağlı olarak ışık yoğunluğu hesaplanır.

3. Ana Çıkış:

a. **FragColor: main()** fonksiyonu **pointLight()** fonksiyonunu çağırarak noktasal ışık hesaplamalarını uygular ve pikselin nihai rengini belirler.

b. İstenirse **directionalLight()** veya **spotLight()** çağrılarak farklı aydınlatma efektleri kullanılabilir.

Bu fragment shader, gerçekçi aydınlatma efektleri oluşturmak için Phong aydınlatma modeli ve Blinn-Phong yansıma modeli kullanır.

Light.vertex ve light.fragment dosyaları ise ışık objemizin oluşturulması için kullanılır. Unutmayınki ışık objesi herhangi bir texture içermez ve rengi beyazdır. Bu nedenle Mesh oluştururken rastgele texture değeri verilir. İleride daha detaylı açıklanacaktır.

CLASS YAPILARI

1. EBO

EBO (Element Buffer Object) sınıfı, OpenGL'de eleman (index) bazlı çizim yapmak için kullanılan bir buffer (tampon) yönetim sınıfıdır. EBO, VAO (Vertex Array Object) ve VBO (Vertex Buffer Object) ile birlikte çalışarak, tekrarlanan verteks verilerini optimize etmeye yardımcı olur.

Sınıfın constructor (**EBO::EBO**) çağrıldığında, önce **glGenBuffers(1, &ID)** fonksiyonu ile OpenGL içinde bir EBO kimliği oluşturulur. Daha sonra **glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID)** ile bu tampon bağlanır, yani OpenGL bu tampon üzerinde işlem yapacağını bilir. Ardından, **glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), indices.data(), GL_STATIC_DRAW)** fonksiyonu ile **indices** vektöründeki veriler GPU belleğine yüklenir. Buradaki **GL_STATIC_DRAW**, bu verilerin sık sık değişmeyeceğini ve yalnızca okunacağını belirtir, yani performans açısından en uygun kullanım belirlenir.

EBO::Bind fonksiyonu, **glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID)** çağrısını yaparak, önceden oluşturulmuş olan EBO nesnesini OpenGL'e bağlar. Eğer bir **VAO (Vertex Array Object) bağlıysa**, bu EBO o VAO ile ilişkilendirilir ve böylece birden fazla nesne ile çalışırken organizasyon kolaylaşır.

Bağlı bir EBO'yu devre dışı bırakmak için **EBO::Unbind** fonksiyonu kullanılır. **glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0)** fonksiyonunu çağırarak, herhangi bir aktif EBO olmadığını OpenGL'e bildirir. Bu, yanlışlıkla başka bir işlemde EBO'nun kullanılmasını önlemek için gereklidir.

Son olarak, **EBO::Delete** fonksiyonu, **glDeleteBuffers(1, &ID)** çağrısını yaparak EBO'yu GPU belleğinden kaldırır. OpenGL'de tamponlar manuel olarak silinmezse, bellek sızıntılarına neden olabilir. Bu nedenle, bir EBO artık kullanılmayacaksa silinmesi performans açısından önemlidir.

2. VBO

VBO (Vertex Buffer Object) sınıfı, OpenGL'de **verteks verilerini** GPU belleğine aktarmak ve çizim işlemlerinde kullanmak için oluşturulmuştur. VBO, **modelin konum, renk, normal ve doku koordinatları gibi bilgilerini** içeren verteksleri depolayarak, CPU'dan GPU'ya veri aktarımını optimize eder. Bu sayede grafik işlemleri **daha hızlı ve verimli** bir şekilde gerçekleştirilir.

Sınıfın constructor (**VBO::VBO**) çağrıldığında, **glGenBuffers(1, &ID)** fonksiyonu ile yeni bir **VBO nesnesi** oluşturulur ve OpenGL tarafından tanımlanan **ID** değişkenine atanır. Daha sonra **glBindBuffer(GL_ARRAY_BUFFER, ID)** fonksiyonu ile bu tampon bağlanır, yani OpenGL **bu VBO üzerinde işlem yapacağını** bilir. **glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), vertices.data(), GL_STATIC_DRAW)** fonksiyonu, **vertices** adlı verteks vektöründeki tüm verileri GPU belleğine kopyalar. **GL_STATIC_DRAW**, bu verilerin **sık sık değişmeyeceğini ve yalnızca okunacağını** belirtir. Bu, VBO'nun **performans açısından daha verimli kullanılmasını** sağlar.

VBO::Bind fonksiyonu, **glBindBuffer(GL_ARRAY_BUFFER, ID)** çağrısını yaparak önceden oluşturulmuş olan **VBO nesnesini bağlar**. Eğer bir **VAO (Vertex Array Object) bağlıysa**, bu **VBO doğrudan o VAO ile ilişkilendirilir**. Böylece, birden fazla nesneyle çalışırken **verteks verilerinin yönetimi kolaylaşır**.

Bağlı bir VBO'yu devre dışı bırakmak için **VBO::Unbind** fonksiyonu kullanılır. **glBindBuffer(GL_ARRAY_BUFFER, 0)** fonksiyonu çağrılarak, OpenGL'e artık **hiçbir VBO'nun bağlı olmadığı** belirtilir. Bu, yanlışlıkla başka işlemlerde **yanlış VBO'nun kullanılmasını engellemek** için gereklidir.

Son olarak, **VBO::Delete** fonksiyonu, **glDeleteBuffers(1, &ID)** çağrısını yaparak **VBO'yu GPU belleğinden kaldırır**. OpenGL'de tampon nesneleri **manuel olarak silinmezse**, gereksiz bellek tüketimine neden olabilir. Bu nedenle, bir VBO artık kullanılmayacaksa, **performans ve bellek yönetimi açısından silinmesi gereklidir**.

3. VAO

VAO (Vertex Array Object) sınıfı, OpenGL'de **verteks verilerini yönetmek ve bağlamak için** kullanılır. VAO, birden fazla **VBO (Vertex Buffer Object)** ve bu VBO'lara ait **nitelikleri (attributes)** organize ederek **GPU üzerindeki veri yapısının düzenli olmasını sağlar**. VAO kullanımı, **birden fazla nesnenin çizim işlemlerini hızlandırır ve yönetimini kolaylaştırır**.

Sınıfın constructor (**VAO::VAO**) çağrıldığında, **glGenVertexArrays(1, &ID)** fonksiyonu ile yeni bir **VAO nesnesi oluşturulur** ve OpenGL tarafından **ID** değişkenine atanır. Bu ID, OpenGL tarafından oluşturulan **benzersiz bir tanımlayıcıdır** ve bu VAO'nun diğerlerinden ayrılmasını sağlar.

VAO::LinkAttrib fonksiyonu, bir **VBO'yu VAO ile ilişkilendirerek** belirli bir **dizilimi (layout)** tanımlar. İlk olarak, **VBO.Bind()** çağrısı ile **bağlanacak VBO aktif hale getirilir**. Daha sonra **glVertexAttribPointer** fonksiyonu ile **VBO'nun içeriği nasıl yorumlanacaksa o şekilde yapılandırılır**. Buradaki parametreler şu şekildedir:

- **layout**: Verteks verisinin hangi konumda tutulacağını belirtir (örneğin, **konum, renk, normal veya doku koordinatları** olabilir).
- **dimension**: Bu verinin **kaç bileşenden oluştuğunu** belirtir (örneğin, 3D konum verileri için 3, renkler için 3 veya 4 olabilir).
- **type**: Verinin hangi **türde** (GL_FLOAT, GL_INT, vb.) saklandığını gösterir.

- **size**: Bir verteksin toplam boyutunu belirtir (yani, bir verteksin tüm bileşenlerinin hafızada kapladığı alan).
- **offset**: VBO içindeki verinin hangi **konumdan itibaren okunacağını** belirtir.

Bu fonksiyon çağrıldıktan sonra, **glEnableVertexAttribArray(layout)** ile ilgili **nitelik (attribute) etkinleştirilir**. Bu işlem tamamlandıktan sonra, **VBO.Unbind()** çağrılarak **bağlı olan VBO serbest bırakılır**, böylece yanlışlıkla başka işlemler tarafından değiştirilmesi önlenir.

VAO::Bind fonksiyonu, **glBindVertexArray(ID)** çağrısını yaparak **bu VAO'yu aktif hale getirir**. Bu sayede, **bağlı olan VBO'lar ve nitelikler (attributes) tekrar tekrar ayarlamaya gerek kalmadan** OpenGL tarafından hatırlanır ve kullanılır.

VAO::Unbind fonksiyonu, **glBindVertexArray(0)** çağrısını yaparak **aktif VAO'yu devre dışı bırakır**. Bu, **diğer OpenGL işlemlerinde yanlışlıkla bu VAO'nun kullanılmasını önlemek için gereklidir**.

Son olarak, **VAO::Delete** fonksiyonu, **glDeleteVertexArrays(1, &ID)** fonksiyonunu çağırarak **bu VAO nesnesini OpenGL'den siler**. VAO'lar GPU belleğinde gereksiz yer kaplamaması için **işleri bittiğinde manuel olarak silinmelidir**.

4. shaderClass

Shader sınıfı, OpenGL'de **vertex ve fragment shaderlarını oluşturup yönetmek için** kullanılır. Shaderlar, GPU üzerinde çalışan küçük programlardır ve **grafik işlemlerini yöneterek** nesnelerin nasıl çizileceğini belirler. Bu sınıf, **shader dosyalarını yükleyip derleyerek bir shader programı oluşturur ve yönetir**.

get_file_contents fonksiyonu, verilen dosyanın içeriğini okuyarak bir **std::string** olarak döndürür. Fonksiyon, dosyayı **binary modunda açar** ve içeriği okumak için sonuna giderek toplam uzunluğunu belirler. Daha sonra dosyanın başına geri dönerek **veriyi string değişkenine okur** ve dosyayı kapatır. Eğer dosya açılamazsa, **hata fırlatır**.

Shader::Shader constructor fonksiyonu, verilen **vertex ve fragment shader dosyalarını okuyarak** OpenGL'de bir shader programı oluşturur. İlk olarak **dosya içeriklerini get_file_contents ile okur** ve C dilinde kullanılabilir hale getirmek için **c_str()** ile işaretçiye çevirir. Ardından **glCreateShader** fonksiyonunu kullanarak bir **vertex shader nesnesi oluşturur**, kaynak kodunu ona bağlar ve **glCompileShader ile derler**. Derleme sırasında **compileErrors fonksiyonu çağrılarak hata kontrolü yapılır**.

Vertex shader tamamlandıktan sonra **aynı işlemler fragment shader için de tekrarlanır**. Fragment shader da oluşturulup derlendikten sonra, **glCreateProgram** fonksiyonu ile bir shader programı oluşturulur ve **vertex ve fragment shaderları bu programa bağlanır**. Daha sonra **glLinkProgram ile tüm shaderlar birleştirilerek çalıştırılabilir hale getirilir**. Programın başarılı bir şekilde oluşturulup oluşturulmadığını anlamak için **compileErrors fonksiyonu çağrılarak hata kontrolü yapılır**.

Shader programı başarıyla oluşturulduktan sonra, **artık gereksiz hale gelen vertex ve fragment shader nesneleri silinir**. Shader programı, **GPU üzerinde çalıştırılacak şekilde hazır hale gelir** ve Shader nesnesinin **ID değişkeninde saklanır**.

Shader::Activate fonksiyonu, oluşturulan shader programını **aktif hale getirmek için** kullanılır. **glUseProgram(ID) fonksiyonu çağrılarak**, OpenGL'in bu shader programını kullanması sağlanır.

Shader::Delete fonksiyonu, shader programını **silerek GPU belleğinden kaldırmak için** kullanılır. **glDeleteProgram(ID)** fonksiyonunu çağırarak, bellekte yer kaplayan shader programını tamamen kaldırır.

Shader::compileErrors fonksiyonu, **vertex, fragment ve program shaderlarının derlenme ve bağlanma hatalarını kontrol etmek için** kullanılır. Shader türüne bağlı olarak, OpenGL fonksiyonlarıyla **derleme veya bağlanma durumu sorgulanır**. Eğer derleme veya bağlanma başarısız olursa, **hata mesajı alınıp ekrana yazdırılır**.

Bu sınıfın amacı, **OpenGL uygulamalarında shader yönetimini kolaylaştırarak, shader oluşturma, bağlama ve hata kontrolünü tek bir yapıda toplamak** ve böylece kodun okunabilirliğini artırmaktır.

5. Camera

Camera sınıfı, OpenGL'de **3D sahnede bir kamera oluşturup yönetmek için** kullanılır. Kamera, sahnedeki nesneleri nasıl gördüğümüzü belirler ve **pozisyon, yön, hız ve kullanıcı girişlerini işleyerek** görüntüyü günceller.

Camera::Camera constructor fonksiyonu, kameranın başlangıç genişliğini, yüksekliğini ve pozisyonunu ayarlamak için kullanılır. **Parametre olarak aldığı width ve height değerlerini sınıf değişkenlerine atar ve kameranın başlangıç konumunu belirler**.

Camera::updateMatrix fonksiyonu, kamera matrisini güncellemek için kullanılır. İlk olarak, **görünüm (view) ve projeksiyon (projection) matrisleri başlatılır**. **glm::lookAt fonksiyonu ile kamera, belirlenen yön ve konum doğrultusunda bakacak şekilde ayarlanır**. Daha sonra, **glm::perspective fonksiyonu kullanılarak kamera perspektifi belirlenir**. Bu işlem, **bakış açısı (FOV), ekran oranı ve yakın-uzak düzlemleri kullanarak bir projeksiyon matrisi oluşturur**. Son olarak, **kamera matrisi hesaplanarak saklanır**.

Camera::Matrix fonksiyonu, shader programına kamera matrisini göndererek görüntünün güncellenmesini sağlar. **glUniformMatrix4fv fonksiyonu ile hesaplanan kamera matrisi, shader programında kullanılmak üzere GPU'ya aktarılır**.

Camera::Inputs fonksiyonu, klavye girişlerini işleyerek **kameranın hareketini ve dönüşünü sağlar**. **W, A, S, D tuşlarıyla öne, sola, geriye ve sağa hareket edebilir**. **Boşluk (SPACE) tuşu yukarı hareket ettirirken, CTRL tuşu aşağı hareket ettirir**. **Shift tuşu basılı tutulduğunda hareket hızı artırılır, bırakıldığında normale döner**.

Kameranın yönünü değiştirmek için, **Q ve E tuşları yatay eksende dönüş (yaw) yapmayı sağlar**. **R ve F tuşları ise dikey eksende dönüş (pitch) yaparak kamerayı yukarı veya aşağı döndürür**. **Pitch açısı 89° ve -89° sınırları içinde tutulur** çünkü daha büyük açılarda kamera ters dönebilir.

Son olarak, **kameranın yön vektörü (Orientation), güncellenen yaw ve pitch değerlerine göre hesaplanır**. **glm::normalize ile vektör normalleştirilerek**, yön kaymasını önleyen düzgün hareket sağlanır.

Bu Camera sınıfı, **birinci şahıs (FPS) kamera kontrolü veya serbest kamera hareketi gerektiren OpenGL projelerinde kullanılabilecek temel bir yapı sunar**.

6. Texture

Texture sınıfı, OpenGL'de bir **doku** (texture) yüklemek ve yönetmek için kullanılan bir yapıdır. Bu sınıf, doku yükleme, bağlama, serbest bırakma ve silme işlemlerini kapsar.

Texture::Texture constructor fonksiyonu, verilen **görüntü dosyasını (image)** OpenGL dokusu olarak yükler ve bu dokuyu bir doku birimine (texture unit) atar. İlk olarak, **STB image kütüphanesini kullanarak görüntüyü yükler ve boyutları ile renk kanal sayısını elde eder**. Daha sonra, **OpenGL'de bir doku nesnesi oluşturur**. Bu nesne, **belirtilen doku birimine (slot) atanır ve bağlanır**. Ardından, **dokunun minifikasyon (küçültme) ve magnifikasyon (büyütme) işlemleri için filtreleme algoritmaları belirlenir**. **Dokunun, s-dönüşü (horizontal wrap) ve t-dönüşü (vertical wrap) parametreleri** de belirlenir, yani dokunun nasıl tekrarlanacağı ya da sınırlarının nasıl işleneceği ayarlanır. **STB image'dan alınan görüntü verisi OpenGL dokusuna yüklenir** ve mipmap'ler oluşturulur. Son olarak, **görüntü verisi serbest bırakılır çünkü artık OpenGL dokusu içinde saklanmaktadır**.

Texture::texUnit fonksiyonu, **shader programına doku birimi bilgisi** gönderir. Bu, shader'ın doğru doku birimini kullanabilmesi için önemlidir. **Uniform değişkeninin konumu bulunur**, ardından **shader aktif hale getirilir** ve **doku birimi ile ilişkilendirilir**.

Texture::Bind fonksiyonu, doku nesnesini aktif hale getirir ve bağlar. Bu, OpenGL'in bu dokuyu kullanmasını sağlar. **Doku birimi (unit)**, önceden belirlenen bir birim üzerinde doku bağlanır.

Texture::Unbind fonksiyonu, **doku nesnesini serbest bırakır** ve bağlamadan çıkartır. Bu, yanlışlıkla dokunun değiştirilmesini engeller.

Texture::Delete fonksiyonu, **OpenGL dokusunu siler** ve ilgili doku nesnesinin kaynağını serbest bırakır. Bu, bellek yönetimi açısından önemlidir, çünkü kullanılmayan dokuların bellekte kalması istenmez.

Bu sınıf, OpenGL projelerinde **dokuları yüklemek, yönetmek ve kullanmak için temel bir araç** sağlar, özellikle 3D grafiklerde doku haritalama ve doku efektleri için çok kullanışlıdır.

7. Mesh

Mesh sınıfı, 3D modelin tüm **düğüm (vertex)**, **indeks** ve **doku (texture)** bilgilerini bir arada tutar ve bu verileri OpenGL'e aktararak modelin çizilmesini sağlar. **Mesh** sınıfı, **vertex** verilerini, **indeksleri** ve **dokuları** alarak bir modelin tam anlamıyla çizilmesi için gerekli olan her şeyi bir araya getirir.

Mesh::Mesh kurucu fonksiyonu, üç ana bileşeni alır: **vertex'ler**, **indeksler** ve **dokular**. Bu veriler daha sonra OpenGL'e aktarılır. İlk olarak, **VBO (Vertex Buffer Object)** ve **EBO (Element Buffer Object)** nesneleri oluşturulur ve sırasıyla **vertex** verileri ve **indeksler** bunlara yüklenir. **VAO (Vertex Array Object)**, **VBO** ve **EBO**'yu bağlayarak, bu nesneler arasındaki ilişkiyi kurar. Bu bağlama işleminden sonra, **VAO'ya (meshVAO) VBO'nun vertex bilgileri**, yani **konum, renk, normal, ve doku koordinatları** gibi atributlar bağlanır. Her bir atribut türü, **LinkAttrib** fonksiyonu ile bağlanır ve doğru şekilde OpenGL'e aktarılır. Son olarak, **meshVAO**, **VBO** ve **EBO** bağlamadan çözülür, böylece veriler üzerinde yanlışlıkla değişiklik yapılmasının önüne geçilir.

Mesh::Draw fonksiyonu, modelin çizilmesi için çağrılır. İlk olarak, **shader programı aktif hale getirilir** ve **meshVAO** bağlanır. Ardından, modeldeki **dokular** döngüyle kontrol edilir. Her bir **doku** türü (diffuse, specular vb.) için doğru doku birimi atanır ve dokular bağlanır. Dokuların hangi türde olduğunu belirlemek için, doku

türüne göre bir numara oluşturulur. Daha sonra, **kamera bilgileri** shader'a aktarılır ve **kamera matrisi** güncellenir. Son olarak, **glDrawElements** fonksiyonu ile modelin **üçgenler** halinde çizimi yapılır.

Bu sınıf, **3D modellerin** doğru şekilde çizilmesi ve yönetilmesi için temel işlevleri sağlar ve **OpenGL** ile etkileşimde bulunarak, verimli bir şekilde **vertex, indeks ve doku verilerini** yönetir. Mesh sınıfı, özellikle **doku kaplaması (texturing)** ve **gelişmiş model yönetimi** gibi işlemler için oldukça kullanışlıdır.

MAIN FONKSIYONU

main() fonksiyonu, OpenGL render ortamını kurmak, kaynakları yüklemek, kullanıcı girişlerini işlemek ve çeşitli nesnelerden oluşan bir 3D sahne çizmek için sorumludur. Bu fonksiyon, pencereyi başlatmak, OpenGL'i yapılandırmak, shader'ları yüklemek, mesh'leri hazırlamak ve nesneleri çizmek için bir dizi adım takip eder. Aynı zamanda kullanıcı girişlerini işleyerek hareket ve kamera kontrolünü sağlar. İşte main() fonksiyonunun detaylı bir açıklaması:

1. GLFW Başlatılması

```
glfwInit();
```

İlk olarak, GLFW başlatılır. GLFW, pencereyi oluşturmak ve OpenGL ile etkileşim sağlamak için kullanılan bir kütüphanedir. Bu fonksiyon, uygulamanın OpenGL işlemlerine başlamadan önce gerekli ortamı hazırlar.

2. Pencere Oluşturma

```
GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Bilgisayar Grafikleri - Proje 1",
NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

Burada, belirtilen genişlik (WIDTH) ve yükseklik (HEIGHT) ile bir pencere oluşturuluyor. Eğer pencere oluşturulamazsa, hata mesajı yazdırılır ve program sonlandırılır. Eğer başarılıysa, pencereyi aktif hale getirmek için glfwMakeContextCurrent(window) çağrılır.

3. GLAD Yükleyicisinin Başlatılması

```
gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);
```

GLAD, OpenGL fonksiyonlarını yüklemek için kullanılan bir kütüphanedir. gladLoadGLLoader fonksiyonu, OpenGL işlevlerini doğru şekilde yükler.

4. Viewport ve Yedek Ekran Boyutlarının Alınması

```
int fbSizeX, fbSizeY;
glfwGetFramebufferSize(window, &fbSizeX, &fbSizeY);
glViewport(0, 0, fbSizeX, fbSizeY);
```

Burada, pencere boyutları (fbSizeX ve fbSizeY) alınır ve glViewport fonksiyonu ile ekranın çözünürlüğü belirlenir.

5. Shader Yüklemesi

```
Shader shaderProgram("Shaders/vertex.shader", "Shaders/fragment.shader");
```

Shader sınıfı, vertex ve fragment shader'larını yükler. Burada, Shaders/vertex.shader ve Shaders/fragment.shader dosyaları yükleniyor. Bu shader'lar, sahne nesnelerinin nasıl işleneceğini belirler.

6. Mesh ve Texture Yüklemeleri

Bu kısımda, her bir nesne için gerekli vertex, indeks ve doku verileri yüklenir:

```
Texture coalTextures[]{Texture("Resources/coal.png", "diffuse", 0, GL_RGB,
GL_UNSIGNED_BYTE)};
std::vector <Vertex> coalVerts(coalVertices, coalVertices + sizeof(coalVertices) / sizeof(Vertex));
std::vector <Texture> coalTex(coalTextures, coalTextures + sizeof(coalTextures) / sizeof(Texture));
Mesh coalCube(coalVerts, cubeInd, coalTex);
```

Burada, örnek olarak "coal" (kömür) adlı bir nesne oluşturuluyor. İlgili dokular (coal.png), vertex ve indeks verileriyle birlikte Mesh sınıfı kullanılarak yükleniyor.

7. Işık ve Işık Shader'ının Yüklenmesi

```
Shader lightShader("Shaders/light.vertex", "Shaders/light.fragment");
```

Bu, ışık nesnesini çizmek için kullanılacak shader'ı yükler.

8. Işık ve Nesne Modellerinin Oluşturulması

```
glm::mat4 lightModel = glm::mat4(1.0f);
lightModel = glm::translate(lightModel, lightPos);
```

Burada, ışık ve diğer nesneler için model matrisleri oluşturulur ve gerekli konumlara yerleştirilir.

9. Derinlik Testinin Etkinleştirilmesi

```
glEnable(GL_DEPTH_TEST);
```

Derinlik testi, sahnedeki nesnelerin doğru sıralanması için gereklidir, bu yüzden burada etkinleştirilmiştir.

10. Kamera Oluşturulması ve Kullanıcı Girişlerinin İşlenmesi

Camera camera(WIDTH, HEIGHT, glm::vec3(0.0f, 0.3f, 2.0f));

Bir kamera nesnesi oluşturulur ve kullanıcıdan gelen girişler işlenerek kameranın pozisyonu ve yönü güncellenir.

11. Ana Render Döngüsü

```
while (!glfwWindowShouldClose(window))
{
    glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    camera.Inputs(window);
    camera.updateMatrix(45.0f, 0.1f, 100.0f);

    coalCube.Draw(shaderProgram, camera);
    diamondCube.Draw(shaderProgram, camera);
    ...
    light.Draw(lightShader, camera);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Bu döngü, pencere kapanana kadar devam eder. Unumayın, saydam nesneler (cam vb.) çizilirken Blend aktive edilmelidir. Her döngüde:

- Arka plan rengi ayarlanır.
- Kamera girişleri işlenir ve kamera matrisi güncellenir.
- Sahne nesneleri çizilir (örneğin, coalCube.Draw).
- Işık kaynağı çizilir.
- glfwSwapBuffers ile ekran yenilenir ve glfwPollEvents ile olaylar işlenir.

12. Temizlik ve Kaynakların Serbest Bırakılması

```
shaderProgram.Delete();
lightShader.Delete();
glfwDestroyWindow(window);
glfwTerminate();
```

Program sonunda oluşturulan shader ve pencere temizlenir, GLFW sonlandırılır.