Başkent Üniversitesi

Mühendislik Fakültesi

Bilgisayar Mühendisliği Bölümü

BIL547- Bilgisayar Grafiği

Proje Ödevi

Geometry Survival

Doğa Sarp Sezer

22410004

Ders Sorumlusu: Dr. Oğul Göçmen

Proje Genel Bakışı

Proje Konusu

Geometry Survival bir hayatta kalma oyunudur. Oyucunun yenilmeden hayatta kaldığı süreye göre bir skor sistemi benimsemiştir. Geometrik şekillerden oluşan oyunda aynı zamanda küçük bir OpenGL menüsü de bulunmaktadır.

Teknolojiler

OpenGL Grafik yordamlarıyla hazırlanan oyunda, olabildiğince günümüz standartlarında kullanılan oyun motorlarının temelinde yatan teknolojiler tekrardan yaratılmaya çalışılmıştır. Bu sayede ortaya modüler ve ilerde daha ileri götürülebilecek bir proje çıkmıştır.

Kullanılan Konseptler

Projede birbirinden farklı güncel ve eski oyun teknolojilerine yer verilmiştir. Bu kullanılan konseptler, şu an daha farklı biçimlerde uygulansa da aslında temelinde bu projedekine benzemektedir.

Debug Sistemi

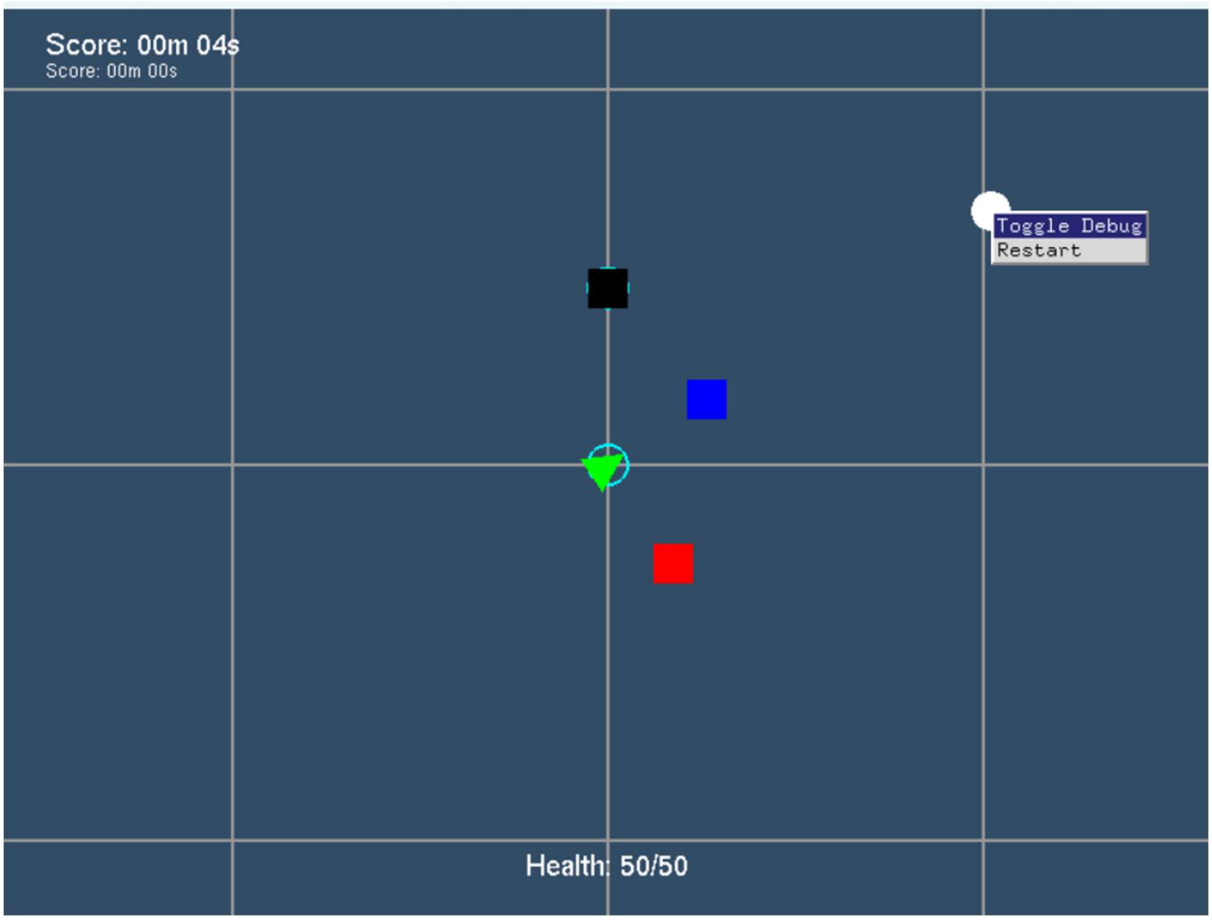
Geliştirme yaparken, geliştiricilerin hataları anlayabilmesi veya yaratılan ortamın görselleştirilmesi için Debug yöntemleri oldukça popülerdir. Projede bu sebepten ötürü bir debug sistemi yazılmıştır.

Debug sistemi, sahneye çizilecek debug objelerini kaydeder ve debug aksiyonunun aktif olması durumunda bu objeleri tuttuğu veriler eşliğinde sahneye çizer. Debug adı altında bu sistemin toparlanmasının en büyük avantajı tek tuşla bu ekstra işlemlerin kapatılabiliyor olmasıdır.

```
SimpleCharacter* playerForwardDebug = new SimpleCharacter(SimpleGeo(SQUARE, blue));
SimpleCharacter* playerRightDebug = new SimpleCharacter(SimpleGeo(SQUARE, red));
SimpleCharacter* inputDebug = new SimpleCharacter(SimpleGeo(CIRCLE, white));

playerForwardDebug->transform.SetPosition(transform.position + transform.Forward() * 3);
playerRightDebug->transform.SetPosition(transform.position + transform.Right() * 3);
inputDebug->transform.SetPosition(hitPos);

Debug::Instance().AddDebug(inputDebug);
Debug::Instance().AddDebug(playerForwardDebug);
Debug::Instance().AddDebug(playerRightDebug);
```



Debug sistemi bütün oyun sistemi içerisinden ulaşılabilen bir global olmalıdır. Bu sebepten ötürü sadece bir adet Debug sistemi olacağını garantileyen Singleton Pattern konsepti ile Debug sistemi yazılmıştır.

Çarpışma

Oyunların en temel işlemlerinden çarpışmalar birden fazla şekilde ele alınabiliyor. Bu projede rotasyonların çarpışmaların kalitesini bozmaması ve oyun içi şekillerin basitliği sebebiyle Daire Çarpışma algoritması kullanılmıştır.

```
inline bool CircleCollision(Vector3& p1,float& r1,Vector3& p2, float& r2)
{
    //sqr length is much faster
    float pointDistance = (p1 - p2).SqrLength();
    float minDistanceForCollision = r1 + r2;

    return pointDistance < minDistanceForCollision * minDistanceForCollision;
}
```

Bu algoritma iki dairenin iç içe olmaması için aralarındaki minimum mesafenin en az dairelerin yarı çaplarının toplamları kadar olmak zorundadır mantığından oluşmaktadır.

Bir optimizasyon olarak Vektör uzunlukları alınırken, kökleri alınmadan hesaplama yapılmıştır.

Çarpışmalar oyun içerisinden farklı yerlerde kullanılmaktadır.

1- Mermilerin düşmanları vurabilmesi
2- Düşmanların, oyuncuyu kovalarken bir yandan da birbirlerini itmesi
3- Düşmanların oyuncuya değer oyuncuya zarar verebilmesi

Işın ve Düzlem

Oyunda Işınlar ve Düzlemler çok kritik bir rol oynamaktadır. Normalde Grafik Geliştirmede Işınlar, gerçekçi ışıklandırmadan, çarpışma kontrolüne kadar her yerde kullanılmaktadır.

Geometry Survival oyununda, ışınlar ve düzlemler kullanılarak fare imlecinin oyun düzlemindeki pozisyonları alınır. Böylece oyuncunun ne tarafa bakması gerektiği hesaplanır. Bu işlem oldukça yaygın olan, düzlem üzerine ışın yollayarak kesiştiği yeri bulmak, bir yöntemle yapılır.

Proje geliştirmesinde eklenen Vektör kütüphaneleri yardımıyla iki vektörün noktasal çarpımı sayesinde, belirlenen bir düzleme düşecek ışının mesafesi öğrenilebilir. Bu kullanılacak ışının belirlenmesindeyse, Ekran pozisyonundan dünya pozisyonuna orada da ışına çeviren yardımcı fonksiyonlar kullanılır. Böyle kameradan atılan ışınlar sayesinde, oyun düzleminde farenin yeri bulunur.

```
inline bool RaycastPlane(const Vector3& rayOrigin, const Vector3& rayDir,
    const Vector3& planePoint, const Vector3& planeNormal,
    Vector3& outHitPoint)
{
    float denom = planeNormal.Dot(rayDir);
    if (fabs(denom) < 0.0001f)
        return false;  // Ray is parallel to the plane, since the normal is always 90 to the plane

    float t = (planePoint - rayOrigin).Dot(planeNormal) / denom;
    if (t < 0)
        return false;  // Intersection is behind the ray origin

    outHitPoint = rayOrigin + rayDir * t;
    return true;
}
```

Burada aslında matematiksel bir denklem ile kanıtlanan Düzlemin her noktasının o düzlemin normaline dik olması kuralı sayesinde bulunur.

Zaman

Oyun Geliştirmede Zaman yönetimi oyunun davranışını oldukça etkiler. Her alet ve performans aynı olmayacağından, yazılan bir kodun stabil olarak her yerde aynı çalışması beklenir. Bunu ise zaman yönetimi yapar. Her kare arasında geçen sürenin hesaplanıp, yapılacak belli işlemlerin bu zamana göre oranlanması sayesinde oyundaki bütün işlemler kare bağımsız hale gelir. Her zaman her yerde aynı çalışır.

```
CustomTime& CustomTime::Instance()
{
    static CustomTime instance; // only created once
    return instance;
}

void CustomTime::Init(std::chrono::steady_clock::time_point startFrame)
{
    lastFrame = startFrame;
}

float CustomTime::Update()
{
    auto now = std::chrono::steady_clock::now();
    std::chrono::duration<float> elapsed = now - lastFrame;
    deltaTime = elapsed.count();
    lastFrame = now;
    return deltaTime;
}
```

```
void Bullet::Move()
{
    float deltaTime = CustomTime::Instance().deltaTime;
    float moveAmount = speed * deltaTime;
    totalDistance += moveAmount;

    Vector3 movement = direction.Normalized() * moveAmount;
    transform.Move(movement);

    float scaleT = 1.0f - totalDistance / range;
    transform.SetScale(Vector3(scaleT,scaleT,scaleT));
}
```

Direksiyon Davranışları (Steering Behaviors)

Direksiyon Davranışları yıllardır Oyun Geliştiricileri tarafından geliştirilmiş bir doğal hareket konusudur. Her bir direksiyon davranışı farklı bir şeyi hedefler. Bazısı bir objeyi takip etmeyi, bazısı ise bu projedeki gibi bir yeri aramayı/ulaşmayı hedefler. Böylece sürat'e bağlı yumuşak ve doğal hissettiren hareketler oraya çıkar. Oyundaki hareket eden her cisim bu prensiplere uyarak hareketini gerçekleştirir.

```
//Movement
float horizontalInput = input.GetKeyboardDirectionHorizontal();
float verticalInput = input.GetKeyboardDirectionVertical();
Vector3 forwardInput = transform.Forward() * verticalInput;
Vector3 rightInput = transform.Right() * horizontalInput;
Vector3 dir = (forwardInput + rightInput).Normalized();

Vector3 desiredVelocity = dir * (speed * deltaTime);
Vector3 force = desiredVelocity - velocity;
Vector3 dampingForce = -velocity* damping;

force = force - dampingForce;

velocity = velocity + force.ClampMagnitude(maxForce) * deltaTime;

velocity = velocity.ClampMagnitude(speed);
transform.Move(velocity);
Update();
```

Bileşen Sistemi (Component System)

Oyun motorlarında bileşen sistemler, objelerin birbirinden bileşenler sayesinde ayrılmasını veya ortaklaşmasını hedefler. Bileşenler eklenebilir ve çıkartılabilir. Bu projede bileşenler sabit olarak objelerin içinde bulunur. Önemli verileri içlerinde saklar ve bileşen görevlerini yerine getirirler.

Collider

Çarpışma mekaniği için gerekli verileri tutar. Çarpışma alanını belirler.

```
class Collider
{
public:
    Collider(float radius) : radius(radius) {}

    float GetRadius(float scale)
    {
        return radius * scale;
    }

    void DrawDebug(Vector3 center, Vector3 scale)
    {
        SimpleCharacter* colliderDebug = new SimpleCharacter(SimpleGeo(CIRCLE_BOUNDS, cyan, 1.0f));
        colliderDebug->transform.SetPosition(center);
        colliderDebug->transform.SetScale(scale * (radius * 2));

        Debug::Instance().AddDebug(colliderDebug);
    }

private:
    float radius;
};
```

Transform

Oyundaki her objede bulunan bir bileşendir. Amacı objelerin dünya içindeki konumlarını, rotasyonlarını ve büyüklüklerini ayarlamaktır. OpenGL bu işlemleri matriks işlemleri sayesinde gerçekleştirir. Aslında Transform bileşeni de arka planda bunu yapar ancak ön planda, geliştiricilerin daha rahat bir sistemde çalışabilmesi için belli yöntemlerle bu işlemler saklanır ve daha basit veri yapılarıyla Transform bileşeni kullanılır.

```
class Transform
{
public:
    Transform(Vector3 position = Vector3(0, 0, 0),
        Vector3 rotation = Vector3(0, 0, 0),
        Vector3 scale = Vector3(1.0f, 1.0f, 1.0f));

    Vector3 Forward();
    Vector3 Right();

    void SetPosition(Vector3 newPosition);
    void SetRotation(Vector3 newRotation);
    void SetScale(Vector3 newScale);

    void Move(Vector3 movement);
    void RotateOnY(float angle);
    void UpdateMatrix(); // builds the model matrix
    GLfloat* ToGLMatrix();

    Vector3 position;
    Vector3 rotation;
    Vector3 scale;
private:


    GLfloat matrix[16]; // column-major, used in rendering
};
```

```
void Transform::SetPosition(Vector3 newPos) { position = newPos; UpdateMatrix(); }
void Transform::SetRotation(Vector3 newRot) { rotation = newRot; UpdateMatrix(); }
void Transform::SetScale(Vector3 newScale) { scale = newScale; UpdateMatrix(); }
```

SimpleGeo

SimpleGeo veri yapısı, OpenGL sayesinde çizilebilen bazı primatifleri hızlıcaz çizmek, takip etmek ve saklamak için yapılmıştır. Bu veri yapısı oyunda görseli oluşturulan bütün modellerde vardır. Aynı zamanda Debug sisteminde çizilen veri yapısıdır. Amacı hem geliştirmeyi kolaylaştırmak hem de şekil çizme mantığını tek bir yerde toplamaktır.

```
enum GeoType
{
    TRIANGLE,
    CIRCLE,
    SQUARE,
    CIRCLE_BOUNDS
};

class SimpleGeo {
public:
    SimpleGeo(GeoType geoType,Color color = white,float size = 1.0f);
    void Draw(Transform transform);
    Color color;
private:
    GeoType geoType;
    float size;
};
```