**EKSAMEN**

**SIMULERING OG MODELERING**

**Andreas Holm**

**NOVEMBER 28, 2016**

# Introduction

## Short description

* Game title: Dum II: Last man standing Stan’s last stand
* Platform: Windows/Linux/MacOs
* Game Type: Fast paced Action
* Point of view:  First Person Shooter

## Target audience

This project is an attempt at creating a physics driven first person shooter for the 2016 finals in the subject “Modellering og Simulering” at Buskerud and Vestfold Universty College.

The game is suitable for adults aged 18 and up, as it glorifies violence and nuclear disasters.

## Backstory

Evil zombies have taken over an American nuclear missile silo located on an island in the mediterrainian sea and threatens to annihilate the major metrolpols of the modern world. New York is already gone and more are soon to follow unless one man can infiltrate the island and sabotage the missiles.That man is “Stan”.

Stan’s mission is to INFIL on the island via paradrop and then proceed to fight his way to the misslepad, where he must sabotage the fueling of the missles so that they fly short and land safely in the sea, killing only fish and ocean creatures, which is okay since nobody likes seafood anyway. Be careful so that you do not cut the fuel to much so that you die yourself.

## Game Concept

DUM II is a one player, modern day, first person shooter with a touch of horror elements. It has been designed to be a fast paced, action oriented, shoot’em up.

The player will control the protagonist using keyboard and mouse in order to move the character about on the island. The main mission of the player is to fight his way to the missile silo and sabotage the fueling of the nuclear missiles.

## Target Platform / Minimum Hardware:

**Platform**

*Windows/MacOs/Linux*

**Game Engine**

*Unity 5*

**System requirements**

**Optimal**

OS: Windows 10

CPU: Initial-core i5-4210M 2.6GHz

GPU: NVIDIA GeForce 840M

RAM: 8 GB DDR3 L Memory

**Notes**

This game will be compiled and tested for Microsoft Windows 10.

Builds for Linux and possibly MacOs will also be added to the repository, but I have at the moment no possibility to test and debug them.

## Gameplay

Dum II starts by inserting the protagonist «Stan» into the game world via a paradrop. The player must decide when to deply the parachute, if the parachute is deployed to early, enemies will have a longer periode of time to shoot at the player while he is still in the air, if the parachute is deployed to late «Stan» can take damage and possibly die from the impact with the ground.

From there the player must utilize his AK47 to fight his way into the missle silo complex before the timer runs out and the missles launch.

When the player reach the missile silo he will be promted with how much fuel he want the missile to contain. If the missile fuel is less than 300 kg of fuel the missile will kill the player. If the missle fuel is more than 600 kg the player loses the game because the missle will reach its target destination.

The missile launch timer is synchronized with the gamelevel sun so that the missile launches when the sun is under the horizon.

**Summarized**

* Paradrop onto the island.
* Navigate to the missile silo.
* Avoid/Kill Zombies.
* Sabotage missles.
* Survive missile launch.

## LEVEL



**Fig.1 Game map**

The player starts the game at the paradrop site, marked blue in the map, and must fight his through the zombies to find his way to the missile site, marked red in the map.



**Fig.2 Rocket**

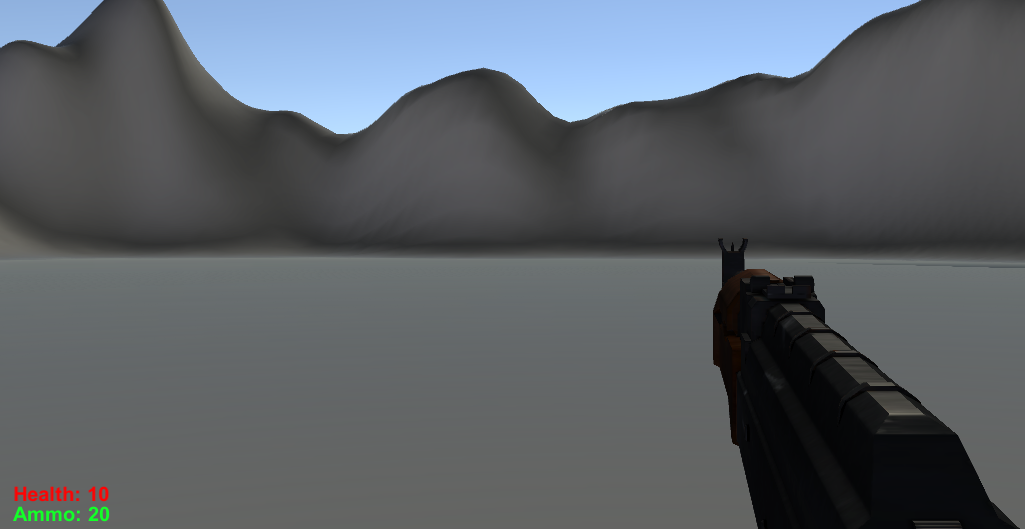


**Fig.3 Standard enemy**

**UI**



**Fig.4 Intro screen.**



**Fig.5 Hud.**



**Fig.6 Aiming.**

****

**Fig.7 Death screen.**

## Mechanics

### Player

**Movement**

To move the character you use the standard «WSAD» first person keyboard keys.

**Looking**

You use the mouse to look around through the eyes of the protagonist.

**Aiming**

Pressing the right mouse button will bring up the AK47 to the center of the scene so that you can aim properly.

**Shooting**

Pressing the left mouse button will fire your weapon.

**Reloading**

When the weapon is out of ammunition, you can press «R» on the keyboard to reload the weapon.

**Deploy Parachute**

While in the air, press «e» on the keyboard to deploy the parachute.

### Enemies

This game has a very limited but effective artificial intelligence simulation in form of a states machine with three different states.

Idle, Move and Shoot.

**Idle**

While the player is out of sight the enemy will stand in his place idle.

**Movement**

If the player is in line of sight, but outside of attack range the enemy will move towards the player so that he can get within attack range.

**Shoot**

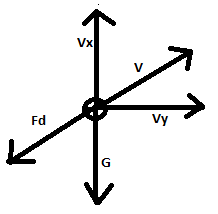
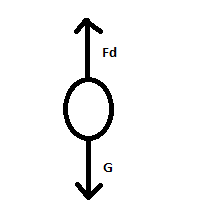
If the player is within line of sight and also within attack range the enemy will proceed to shoot or throw stuff at the player.

# Physics

## Projectile

**Paratrooper Bullet**

**Forces Forces**

****

**Projectile motion with Air Drag**

**Cd is the drag coefficient**

**is the density of air**

**A is the cross-sectional area of the projectile**

**Vx is the x component of the velocity**

**X-dir:**

**dx=m\*ax**

**In Y dir going up:**

**In Y dir going down:**

**So in Y dir:**

**Eulers’ method**

### Solving the Equation in MatLab

**File: Bullet.m**

**clear;clc**

**global C g**

**g=9.81; %m/s^2**

**v0=850; % m/s inital launch speed**

**beta0=30\*pi/180 %initial launch angle in degrees converted to radians**

**m=0.2; % mass of projectile, kg**

**d=0.5; %diameter of spherical projectile, meters**

**Cd=0.5; % assumed**

**rho=1.2041; %density of air, kg/m^3**

**A=pi\*d^2/4**

**C=Cd\*A\*rho/2/m; % drag force constant**

**%% Calculations**

**tmax=5; % Calculate for 5 seconds of flight**

**tspan = [0 tmax];**

**% Initial conditions as [x0, vx0, y0, vy0]**

**IC = [0; V0\*cos(beta0); 0; V0\*sin(beta0)];**

**[t, oput] = ode45(@secondode, tspan, IC);**

**x= oput(:,1)**

**vx=oput(:,2);**

**y=oput(:,3);**

**vy=oput(:,4);**

**figure(1);clf;**

**plot(x,y);**

## Rocket

**(ran out of time)**

**The rocket is only partly finished. Only utilizing Newtons third law on Fn=mve**

### Implementation in Code

**Projectile using Heun’s method as a corrector and Euler as predictor**

### Euler Forward

**This Method takes the time step(h), the current Velocity and Position and uses them to calculate the new Position and the new Velocity, by adding force by a gravity vector and a drag vector which is calculated in «CalculateDrag».**

public static void Euler Forward(

float h,

Vector3 currentPosition,

Vector3 currentVelocity,

out Vector3 newPosition,

out Vector3 newVelocity

)

{

Vector3 accelerationFactor = new Vector3(0f, -9.81f, 0f);

accelerationFactor += BulletPhysics.CalculateDrag(currentVelocity);

Vector3 velocityFactor = currentVelocity;

newPosition = currentPosition + h \* velocityFactor;

newVelocity = currentVelocity + h \* accelerationFactor;

}

### Heun’s method

**The same method but with a corrector implemented, called Heun’s method.**

**We do this by calculating the intermediate value and then the final approximation at the next integration point. Using the Euler’s method as a predictor and Heun’s as a corrector minimizes the truncation error and gets a much more accurate result.**

public static void Heuns(

float h,

Vector3 currentPosition,

Vector3 currentVelocity,

out Vector3 newPosition,

out Vector3 newVelocity)

{

Vector3 accelerationFactorEuler = new Vector3(0f, -9.81f, 0f);

Vector3 accelerationFactorHeun = new Vector3(0f, -9.81f, 0f);

Vector3 velocityFactor = currentVelocity;

Vector3 pos\_E = currentPosition + h \* velocityFactor;

accelerationFactorEuler += BulletPhysics.CalculateDrag(currentVelocity);

Vector3 vel\_E = currentVelocity + h \* accelerationFactorEuler;

Vector3 pos\_H = currentPosition + h \* 0.5f \* (velocityFactor + vel\_E);

accelerationFactorHeun += BulletPhysics.CalculateDrag(vel\_E);

Vector3 vel\_H = currentVelocity + h \* 0.5f \* (accelerationFactorEuler + accelerationFactorHeun);

newPosition = pos\_H;

newVelocity = vel\_H;

}

### CalculateDrag

**This method calculates the drag velocity Vector that acts upon the object.**

**It takes in the mass of the object(m), the drag coefficient(C\_d), the Areal(A) of the object, the fluid density(rho), k is a constant that collects the effects of density, drag and area. After all, that we get the drag vector by multiplying k with v squared and divide them by the total mass of the object. Now we just need to reverse the Drag vector since it works opposite of the velocity of the object.**

public static Vector3 CalculateDrag(Vector3 velocityVec)

{

//F\_drag = k \* v^2 = m \* a

//k = 0.5 \* C\_d \* rho \* A

float m = 0.2f; //kg

float C\_d = 0.5f;

float A = Mathf.PI \* 0.05f \* 0.05f; //m^2

float rho = 1.225f; // kg/m3

float k = 0.5f \* C\_d \* rho \* A;

float vSqr = velocityVec.sqrMagnitude;

float aDrag = (k \* vSqr) / m;

//OBS MUST BE OPPOSITE OF MOVING DIRECTION

Vector3 dragVec = aDrag \* velocityVec.normalized \* -1f;

return dragVec;

}

### Rocket

**RigidRocket and RocketEngine are the main body of the rocket equation, which unfornunatly is still unfinished due to time constraints. The rocket is only modelled after the left side of the rocket equation, namely F = mve. The Rocket produces thrust by exerting mass as in newton’s third law.**

public class RigidRocket: MonoBehaviour {

public float mass; // [Kg]

public Vector3 velocityVector; //[m s^-1]

public Vector3 netForceVector; // N [kg m s^-2]

private List<Vector3> forceVectorList = new List<Vector3>();

public void AddForce(Vector3 forceVector)

{

forceVectorList.Add(forceVector);

Debug.Log("Adding force " + forceVector + " to " + gameObject.name);

}

void UpdatePosition()

{

netForceVector = Vector3.zero;

foreach (Vector3 forceVector in forceVectorList)

{

netForceVector = netForceVector + forceVector;

}

forceVectorList = new List<Vector3>();

Vector3 accelerationVector = netForceVector / mass;

velocityVector += accelerationVector \* Time.deltaTime;

transform.position += velocityVector \* Time.deltaTime;

}

public class RocketEngine : MonoBehaviour {

public float fuelMass; //[kg]

public float maxThrust; //kN [kg m s^-2]

[Range (0,1f)]

public float thrustPercent; //[none]

public Vector3 thrustUnitVector; //[none]

public Vector3 gravity;

private RigidRocket rigidRocket;

private float currentThrust; // N

void Start ()

{

rigidRocket = GetComponent<RigidRocket>();

rigidRocket.mass += fuelMass;

gravity = new Vector3(0f, -9.81f, 0f);

}

float FuelThisUpdate()

{

float exhaustMassFlow;

float effectiveExhaustVelocity;

effectiveExhaustVelocity = 4462f; //[m s^-1] liquid H O

// thrust = massFlow \* exhaustVelocity

//massFlow = thrust / exhaustVelocity

exhaustMassFlow = currentThrust / effectiveExhaustVelocity;

return exhaustMassFlow \* Time.deltaTime; // [kg]

}

void ExertForce()

{

currentThrust = thrustPercent \* maxThrust \* 1000f;

Vector3 thrustVector = thrustUnitVector.normalized \* currentThrust; // N

rigidRocket.AddForce(thrustVector);

rigidRocket.AddForce(gravity);

# 4.References

[1] Grant Palmer. Physics for Game Programmers. Apress, 2005.

[2] Ian Millington, Game Physics Engine Development, Elsevier, 2007

[3] Morgan Kaufmann, Game Physics Second Edition, Elsevier, 2010