Just Duke It – The Game

Abgabe 3
CG23 LU SS04
TU-Wien

Granzer Wolfgang
0026013
E-881
woif@woif.org

Praus Fritz
0025854
E-881
fritz@praus.at

1.Hintergrundstory

Wir schreiben das Jahr 2204. Das Erdölzeitalter neigt sich dem Ende zu. Panik macht sich unter den letzten Motorradfreaks breit. Ohne das schwarze Gold würden die Motorradfahrer dazu verdammt werden, auf Elektromotorräder umzusteigen. Das würde das Ende der harten 4 Takt Einzylinder bedeuten. Kein Prellen, kein Röhren, wenn der Kolben Durst nach Benzin verspürt. Deshalb ist ein Krieg um die letzten Benzinreserven unter den Motorradfahrern ausgebrochen. Nur der schnellste, wendigste und zielsicherste Motorradfahrer ist würdig, die letzten Tropfen Öl zu ergattern.
Also, rauf auf die Bikes und „Frag them all“.

2.Kurzbeschreibung

Da wir einerseits leidenschaftliche Motorradfahrer sind und andererseits gerne First Person Shooter spielen, haben wir uns gefragt, warum wir nicht beides in einem Spiel vereinen sollen. Daher haben wir uns entschlossen ein Spiel zu entwickeln, bei dem es sich um eine Mischung aus Motorradsimulation und 3D Shooter handelt. Man soll daher einerseits in der Lage sein, interaktiv ein Motorrad über ein Terrain zu steuern und andererseits seine „Zielgenauigkeit“ unter Beweis stellen können. Es soll daher möglich sein, aus der Sicht des Motorradsfahrers das Leben seiner Gegner mithilfe von Schusswaffen schwer zu machen. Um für den richtigen Spielespass zu sorgen, haben wir uns außerdem entschlossen unseren „3D Motorradshooter“ als Multiplayerspiel zu konzipieren d.h. es wird möglich sein über TCP/IP gegen andere „reale“ Gegner zu spielen.

3.Spieldesign

Jeder Spieler startet, wie es in einem 3D Shooter üblich ist, mit einer Gesundheit von 100%. Pro gegnerischen Trefferpunkt wird nun die Gesundheit des Spielers verringert bis er schließlich eine Gesundheit von 0 erreicht und von neuem beginnen muss. Derjenige Gegner, der für das Ableben eines Mitspielers verantwortlich ist, erhält einen „Fragpunkt“. Ziel des Spieles ist es, als erstes das so genannte Fraglimit zu ereichen.
Zusätzlich wird es weitere allseits bekannte Powerups wie Amor, Munition und spezielle Powerups geben.

4.Systemvorraussetzungen

Client: Microsoft Windows
1Ghz
128 MB Ram
ATI Radeon oder gleichwertig

Server: Java Runtime Environment 1.4
400 Mhz
128 MB Ram

5.Programmstart

Bei dem Spiel handelt es sich um eine Client – Server Architektur.
Der Server wird z.B. durch

bin/server/jdi_server.exe oder *bin/server/jdi_server-linux*

gestartet. (benötigt selbe Verzeichnisstruktur wie Client, d.h. ganze *.zip* Datei entpacken). Des Weiteren ist ein public access server unter 80.109.42.218 Port 1666 erreichbar (Defaulteinstellung im Konfigurationsfile)
Das Spiel kann mit

bin/justDukeIt.exe

gestartet werden. Alle Einstellungen können über das Menü getroffen werden.

6.Konfiguration und Steuerung

Folgende Parameter können im Spielmenü/jdi.cfg file verändert werden:

Bezeichnung	***Default***	***Beschreibung***
server_ip	80.109.42.218	IP Adresse des Servers
server_port	1666	Port des Servers
playername	hugo	Spielername (max. 7 Zeichen)
duketype	green	Dukefarbe (red od. green)
invert	false	Mausinvert ein/aus
sensitivity	5	Maussensitivität
action_fwd	w	Beschleunigen
action_back	a	Bremsen
action_left	s	Links lenken
action_right	d	Rechts lenken
action_centerview	e	Kamera zentrieren
weapon_2	2	Maschinengewehr auswählen
weapon_7	7	Railgun auswählen
resolution	1	640x480=0 800x600=1 1024x786=2 1280x1024=3
shadow	true	Echtzeitschatten ein/aus
blur	true	Motionblur ein/aus
lensflare	true	Lensflare Effekt ein/aus
fog	true	Nebel ein/aus
sound	true	Sound ein/aus
multitexturing	true	Multitexturing ein/aus
mipmapping	true	Mipmapping ein/aus
texturqual	2	Texturqualität
rendermode	3	0=Immediate 1=Vertex Array 2=Display List 3=Vertex Buffer Objects
fov	90	Fov
lod	false	Levels of Detail ein/aus

Desweiteren sind folgende Funktionstasten belegt:

<F1>: Kameraposition ändern
<F2>: Framerate anzeige ein/aus
<F3>: Wireframe Mode ein/aus
<F4>: Texturqualität ändern
<F5>: Mipmapping ein/aus
<F6>: Rendermode auswählen
<F7>: Levels of Detail ein/aus
<F8>: View Frustum Culling ein/aus
<F9>:
<F10>: Schatten ein/aus
<F11>: Multitexturing ein/aus
<F12>: Exception ☺ Bug in Glut?
<Einfg>: Lensflare ein/aus
<Pos1>: Sound ein/aus
<Bild rauf>: Backface Culling ein/aus
<Bild runter>: Motionblur ein/aus
<Ende>: Cockpit ein/aus
<ESC>: Spiel beenden

7.Features

Derzeit sind 2 Arten von Waffen implementiert: Maschinengewehr und Railgun. Mit dem MG ist es relativ einfach, die gegnerischen Spieler zu bekämpfen. Es bietet jedoch nicht so eine große Durchschlagskraft wie die Railgun. Dafür braucht es jedoch nicht so lange zum Nachladen.
Zu jedem richtigen Shooter gehören **Powerups**. In JDI gibt es folgende:

- kleine Health: Eristoff Ice Flasche
- große Health: Gösser Bier Flasche
- kleine Amor: Motorradhelm
- große Amor: KTM Ledergewand
- MG Munition: Maschinengewehr
- Railgun Munition: Railgun

Diese können (sofern die maximale Aufnahmekapazität nicht erreicht ist) von Spieler aufgenommen werden und sie werden zur leichteren Auffindbarkeit mit einer farbigen Kugel über ihrer Position gekennzeichnet.
Weiters gibt es „**Environment Objekte**“. Dies sind stationäre Objekte, die im Prinzip nur bei der Collision Detection und zur Schaffung einer positiven Aura verwendet werden. (z.B. Gebäude) In der derzeitigen Fassung existieren jedoch noch keine Modelle und deswegen ist das Feature noch nicht aktiv.
Zur weiteren Verbesserung der Atmosphäre wurden mittels FMOD **Sound** Files eingebunden, die z.B. bei Aufnahme eines Items oder beim Frag eines Spielers ertönen. Der Sound wurde dabei vom eigenen Motorrad selbst aufgenommen. ☺
Weiters wurde ein rudimentäres **Physikmodell** entworfen, welches es ermöglicht, über Berge zu springen und den anderen Spielern das Leben

schwer zu machen. Das Modell wird aber auch zur Darstellung und Berechnung des Partikelsystems verwendet.
JDI bietet auch die Möglichkeit **mehrerer Levels**. Dazu muss lediglich im *bin/level.xml* file ein neuer Eintrag erstellt werden. Die Syntax ist dabei selbsterklärend. Der Level wird automatisch gewechselt, sobald das Fraglimit erreicht ist.
Der **Server** wurde zwecks Platformunabhängigkeit in Java implementiert und ist für die korrekte Abwicklung des Spiels zuständig. Weiters verwaltet er die Clients. Der komplette Multiplayer Code ist derzeit, trotz Verwendung von TCP/IP noch nicht sehr fehlerrobust und es können falsche/zerstörte Packete empfangen werden.

8. Implementierung

8.1. Entwurf der Datenstrukturen

Jedes 3D Objekt (z.B. *Duke, Item, Bullet*) ist von der Klasse *AbstractObject* abgeleitet. Diese Klasse ist wiederum von der Klasse *Mass* abgeleitet. Die Klasse *Mass* enthält die Implementierung der Physik. Mithilfe dieser Klasse ist es möglich, eine Geschwindigkeit sowie auf das Objekt einwirkende Kräfte zu simulieren. Die Klasse *Abstractobject* dient zum Laden der 3D Objekte, zur Berechnung der Arrays, zum Erstellen der DisplayLists sowie zur Schattenberechnung. Da bei uns jedes Objekt von den Klassen *Abstractobject* und daher auch von der Klasse *Mass* abgeleitet ist, kann jedes Objekt nicht nur einen Schatten werfen, sondern es kann auch der Einfluss von Kräften (z.B. Schwerkraft) simuliert werden.

Im Headerfile *datastructures.h* befinden sich sämtliche Definitionen unserer verwendet Datenstrukturen.

- *vertex2D*

Implementierung eines 2 dimensionalen Vektors.

- *vertex3D*

Implementierung eines 3 dimensionalen Vektors

- *vertex4D*

Implementierung eines 4 dimensionalen Vektors (besitzt zusätzlich zum vertex3D die homogene Koordinate w)

- *Pos3D*

Diese Datenstruktur repräsentiert eine 3 dimensionale Position im Raum. Zusätzlich zu den Koordinaten *x*,*y* und *z* enthält diese Struktur die Winkel *angleX*, *angleY* und *angleZ*. Mit diesem Winkel können wir ein beliebiges Objekt im Raum positionieren.

- *t3DModel*

Diese Datenstruktur repräsentiert ein 3D Modell. Es enthält einerseits Objekte (STL vector *pObject*) und andererseits Materialien (STL vector *pMaterials*). Die Anzahl der Objekte bzw. Materialien wird mittels *numOfObjects* und *numOfMaterials* festgelegt.

- *t3DObject*

Diese Datenstruktur dient zur Repräsentation eines 3D Objekts. Sie enthält sämtliche Vertices, Normalvektoren und Texturkoordinaten (UV Koordinaten) als Liste abgespeichert. Weiters enthält sie eine Liste der Oberflächen, in der auf die Vertices, Normalvektoren und Texturkoordinaten der einzelnen Punkte verwiesen wird (siehe *tface*).
Des Weiteren enthält die Struktur die Vertexarrays, Normalarrays, Texturarrays und Colorarrays, die in Opengl direkt verwendet werden können. (werden mittels Methode generateArrays in der Klasse AbstractObject erstellt). Außerdem werden sowohl der Displaylistindex sowie das Objektkoordinatensystem zurückgerechnete Lichtposition abgespeichert (wird für die Schattenberechnung benötigt).

- *tFace*

Diese Struktur enthält die Verweise auf die Eckpunkte einer Oberfläche (in unserem Fall immer Dreiecke). Des Weiteren enthält sie die Verweise auf die Eckpunkte der angrenzende Fläche (*neighbourIndices)* sowie ein Flag, ob diese Fläche von der Lichtquelle aus sichtbar ist (wird ebenfalls für Schattenberechnung benötigt). Außerdem wird zusätzlich die Flächengleichung abgespeichert, welche ebenfalls für die Darstellung eines dynamischen Schattens benötigt wird.

- *tPlane*

Enthält die Parameter der Flächengleichung.

8.2. Entwurf eines Kameramodells - animierte Objekte

Die Steuerung der Spielfigur bzw. der Kamera teilt sich in 2 Teile auf:

- Steuerung des Motorrads

Mit Hilfe der Tastatur kann das Motorrad über das Terrain gesteuert werden. Mit den Tasten „beschleunigen“ und „bremsen“ kann die Geschwindigkeit des Motorrads und somit die Position der Kamera verändert werden. Wird keine der beiden Tasten gedrückt, verzögert das Motorrad gleichmäßig (Simulation der Reibung zw. Motorrad und Fahrbahn). Mit Hilfe der Tasten „links“ und „rechts“ wird die Richtung des Motorrads verändert. Je länger einer der beiden Richtungstasten gedrückt wird, umso so mehr legt sich das Motorrad in die Kurve. Da sich der Spieler und somit die Kamera mit dem Motorrad mitbewegen, ergeben sich zusätzlich zur Positionierung 3 Rotationsmöglichkeiten:

- Fahrtrichtung des Motorrades
- Neigungswinkel, welcher durch das „hineinlegen“ des Motorrads beeinflusst wird
- Winkel des Motorrads aufgrund der Unebenheit des Terrains

- Steuerung der Blick/Schussrichtung

Zusätzlich kann der Spieler die Blickrichtung und somit die Richtung der Kamera mit Hilfe der Maus beeinflussen. Somit ergeben sich 2 zusätzliche Rotationsmöglichkeiten. Es ist allerdings zu beachten, dass die Position des Motorrads direkten Einfluss darauf hat. Ändert sich der Neigungswinkel des Motorrads so dreht sich die Kamera mit, auch wenn der Spieler die Maus nicht bewegt. Um die Trägheit des Körpers des Fahrers zu simulieren, wird dieser Einfluss der Richtung des Motorrads etwas verzögert und langsamer durchgeführt. Es entsteht der Eindruck, dass der Fahrer mit dem Terrain teilweise mitgeht und nicht starr am Motorrad sitzt.

Weitere animierte Objekte:

- Powerups

Damit unsere 3D Welt nicht zu starr erscheint drehen sich unsere Powerups gleichmäßig um die vertikale Achse. Dies hat außerdem den Vorteil, dass sie auch bei größerer Entfernung besser erkennbar sind.

- Explosion des Motorrads

Wird ein Spieler „gefragt“, explodiert dabei sein Motorrad. Bei so einer Explosion werden die einzelnen Teile mit Hilfe der implementierten Physik von Zentrum des Motorrads aus in die Luft geschleudert.

- Motorrad Reifen

Die Reifen des Motorrads rotieren geschwindigkeitsabhängig.

- Waffe

Der MG Lauf rotiert abhängig von der Anzahl der Schüsse. Weiters wird die gewählte Waffe immer in Richtung Blickrichtung mitgedreht.

8.3. Objekte / Texturen / Beleuchtung

Die Objekte/Modelle wurden von uns selbst mit 3D Studio Max 6 modelliert und texturiert und in das *.3ds* Format exportiert.

Alle Objekte besitzen Normalvektoren. Diese wurden entweder aus dem 3DS geladen, oder erst im Programm (z.B. Terrain) berechnet. Zusätzlich werden für die meisten Objekte die Nachbarschaftspolygone berechnet. Weiters wurden Texturen im 3D Studio zugewiesen und die Texturkoordinaten gespeichert bzw. berechnet. Die Beleuchtung der Szene erfolgt durch eine „real-world directional light source“, die die Sonne simulieren soll.

8.4. Beschleunigung der Sichtbarkeitsberechnung

Zur Beschleunigung der Sichtbarkeitsdarstellung wurde View Frustum Culling implementiert.

Um ein Viewfrustum Culling der Objekte durchführen zu können, muss zunächst das Blickfeld berechnet werden. Dabei werden zunächst die beiden Matrizen (Modelview- und Projectionmatrix) bestimmt und miteinander multipliziert. Da diese Berechnung in jedem Frame durchgeführt wird und eine Vielzahl von Rechenoperationen notwendig ist, wurde diese vereinfacht (diese Vereinfachung ist allerdings nur möglich, wenn die Projectionmatrix keine Rotation und keine Translation aufweißt). Anschließend werden die einzelnen Flächen des Viewfrustums berechnet.

Für die Culling Tests wurden 2 Methoden für den Test „Punkt in Viewfrustum“ und „Kugel in Viewfrustum“ implementiert.

8.5. Transparenz-Effekte

Einfache Transparenzeffekte wurden z.B. bei der Explosion, beim Railgunschuß, beim Cockpit oder bei der Terrainabgrenzung realisiert. Siehe dazu Punkt 9.

8.6. Experimentieren mit OpenGL

Mithilfe der verschiedenen F Tasten können verschiedene OpenGl Einstellungen verändert werden. Zunächst ist es möglich, verschiedene Rendermodi auszuwählen. Es kann zwischen Immediate Mode (naives glBegin() glEnd() rendern), Vertex Arrays, Display List und Vertex Buffer Object (verfügbar als ARB Extension) gewählt werden. Zu beachten ist, dass bei der Verwendung von VBO, Vertex Arrays nicht mehr funktionieren. Wir sind leider nicht dahinter gekommen, warum dies der Fall ist (Nach dem Anlegen eines VBO mittels glGenBufferARB(…) ist die Darstellung mittels Vertex Arrays falsch bzw. nicht mehr vorhanden). Weiters können verschiedene Textureinstellungen geändert werden. Es besteht die Möglichkeit MipMapping ein bzw. auszuschalten und zwischen Point, bilinear und trilinear Sampling umzuschalten. Außerdem besteht die Möglichkeit, sämtliche Transparenzeffekte, Viewfrustum Culling und LOD ein und auszuschalten. Die Qualität der Grafik kann direkt beobachtet werden. Um die Veränderung der Geschwindigkeit zu beobachten, befindet sich links oben eine FPS Anzeige.

9.Spezialeffekte

9.1. Explosionen

Die Explosionen wurden mithilfe eines Billboarding Systems und mehreren Texturen realisiert. Explodiert ein Spieler bzw. ist ein Einschuss zu sehen, wird ein Rechteck gerendert, welches immer in Blickrichtung Kamera zeigt. Die Texturen werden zeitabhängig aufgetragen bzw. verändert, sodass der Effekt einer kontinuierlichen Animation vorgespielt wird. Weiters werden sie im Blending Mode aufgetragen, sodass die Explosion auch transparent wird und realistischer wirkt.

9.2. Schatten – Schattenvolumen

Zur Berechnung von dynamischen Schatten wurde ein Shadow Volume Algorithmus verwendet. Beim Laden der Objekte werden einmalig pro Fläche die Flächengleichungen berechnet. Diese bleiben während des Programmablaufes konstant. Diese Flächengleichungen werden später für die Sichtbarkeitsberechnung benötigt. Weiters werden zu Beginn für jede Fläche die angrenzenden, benachbarten Flächen bestimmt. All diese Informationen werden für die Schattenberechnung benötigt.
Allerdings ändern sich natürlich die Weltkoordinaten der Objekte, die Schatten werfen. Dadurch würden sich auch die Flächengleichungen ändern.

Um nicht bei jedem Frame alle Flächengleichungen neu zu berechen, wird die Lichtquellenposition in Objektkoordinaten rück transformiert.
Dadurch wird einiges an Rechenaufwand gespart. Als zusätzliche Schwierigkeit trat bei uns das Problem auf, dass sich die Reifen der Motorräder und die Waffe des Fahrers zusätzlich zu den normalen Transformationen noch um die eigene Achse drehen. Daher werden für diese Objekte die Lichtquellen extra berechnet.
Anschließend wird bestimmt, welche Flächen sichtbar sind. Diese Information ist notwendig um die Silhouette der Objekte zu berechnen.
Ist eine Fläche sichtbar und deren Nachbarfläche hingegen unsichtbar, handelt es sich bei der Berührungskante um eine Silhouettenkante (die Umkehrung gilt natürlich auch). Von diesen Kanten aus, wird nun das Schattenvolumen in die „Unendlichkeit“ gezeichnet. Um nun zu bestimmen, ob sich ein Bildpunkt im oder außerhalb des Schattenvolumen befindet, wurde der so genannte „Z- Pass“ Algorithmus verwendet. Im ersten Pass dieses Algorithmus wurde bei allen Frontfaces des Schattenvolumens der Stencil Buffer um 1 erhöht. Im zweiten Pass hingegen wurde bei einer Backface der Stencil Buffer erniedrigt.
Nach den beiden Passes wurde nun ein graues Rechteck über den Framebuffer geblendet. Neben dem Tiefentest wurde auch der Stencil Test aktiviert. War der Tiefentest erfolgreich und ist der Stencilwert im entsprechenden Fragment ungleich 0, wurde dieses Fragment überblendet, da dieses Fragment im Schatten liegt.
Probleme:
Leider treten bei beim Schatten mancher Objekte Artefakte auf. Der Grund liegt wahrscheinlich darin, dass manche Objekte Fehler enthalten (z.B. Löcher in den Objekten). Uns war es allerdings leider nicht möglich bis zur Abgabe alle Fehler auszubessern.

9.3. Lens Flare

Um die Spiegelung der Sonne in der Kamera zu simulieren wurde ein 3D Lensflare Effekt implementiert. Ein realistischer Lensflare Effekt soll nur dann zu sehen sein, wenn die Sonne im Blickfeld des Betrachters liegt und sie nicht von anderen Objekten der 3D Welt verdeckt wird. Um zu überprüfen, ob sich die Sonne im Blickfeld befindet, wird das bereits implementierte Viewfrustum Culling verwendet. Die Überprüfung, ob andere Objekte die Sonne verdecken, wird mithilfe des Z Buffers durchgeführt. Zuerst wird die Viewport Position der Sonne mithilfe des Kommandos gluProject bestimmt. Als nächstes wird nun der momentane Wert des Z Buffers an dieser Position gelesen und mit dem Z Wert (aus gluProject) der Sonne verglichen. Ist der Z Wert der Sonne größer, ist diese verdeckt. Nach der Sichtbarkeitsberechnung werden nun die verschiedenen Texturen des Lensflares gerendert. An der Position der Sonne wird eine große „Leuchttextur“ gezeichnet. Entlang der Gerade zwischen Sonne und Kameraposition werden mehrere „Halos“ gerendert. Zu beachten ist, dass die momentane Kameraposition und Lage berücksichtigt werden muss, damit die Texturen immer in Richtung Kamera zeigen.

9.4. Statische Levels of Detail

Wir haben extreme Performanceprobleme bekommen, als wir Multitexturing bzw. Fog implementiert haben (ca. 50% Performanceeinbußen) und noch nicht mit vertex buffer objects gearbeitet haben. Deshalb wurden statische LOD realisiert. Abhängig von der Kameraposition – d.h. Entfernung Kamera Quadrant - wird für jeden Quadranten der aktuelle Detaillevel gewählt (alle, mittlere, geringe Details). Dieser Detaillevel wird dann auf alle in den Quadranten befindlichen Objekte angewendet. Diese Maßnahme bringt bei Rendermode Displaylist auf einem 3.06Ghz Pentium ca. 40 fps. Im Rendermode VBO ist jedoch kein Performanceverlust zwischen LOD ein/aus mehr festzustellen. Die derzeitige Implementation von LOD ist extrem auf Performancesteigerung ausgerichtet, d.h. bei genauerem Betrachten sind die Übergänge zwischen den Detailstufen sichtbar.

9.5. Motion Blur

Der Motionblur wurde mithilfe des Accumulation Buffers realisiert.
Die Funktion des Blur Effekts läuft in 2 Schritten ab.

1. Den aktuellen Framebufferinhalt in den Accumulation Buffer blenden
2. Aus dem Accumulation Buffer in den Frame Buffer blenden

Dadurch entsteht ein verschwommener Effekt, der die Geschwindigkeit des Motorrades simuliert.

9.6. Cockpit - Radar

Im Cockpit werden die aktuell gewählte Waffe sowie ein Radar angezeigt. Beide Features werden im Blendmode über alle Objekte gerendert. Beim Radar werden die aktuellen Positionen der Items auf ein 2D Abbild projiziert und dann über der Karte des Levels angezeigt. Weiters werden der eigene Spieler sowie die Blickrichtung projiziert.

9.7. Railgun

Der Railgunschuss wurde mit einem Partikelsystem implementiert: Der Beam ist ein Rechteck, welches immer normal zur Kamera steht. Der spiralförmige Trail besteht aus Partikeln, welche in einer Sinuskurve vom Start- zum Endpunkt gerendert werden. Alle Partikel haben einen Alphawert, der über die Zeit verringert wird.

10. Libraries

Es wurden folgende, von der CG-Homepage empfohlenen Libraries, verwendet:
3DS Loader: Loader von www.gametutorials.com
Coldet: Collision Detection Library. Relativ einfache Library, die jedoch ein paar Einschränkungen mit sich bringt, aber für unsere Zwecke ausreichend ist.
FMOD: Gute Library mit 3D Sound
Glut32

Xerces: XML Parsing Library, zur Verarbeitung der level.xml Datei, Source Code nicht in Abgabefile inkludiert!!!

Tutorials:
http://www.gametutorials.com
http://nehe.gamedev.net/
http://submarine.org.uk/