Mosquito Experiment Dokumentation

Inhalt

Ich habe versucht die Dokumentation möglichst vollständig, jedoch auch aufs wesentliche reduziert projektspezifisch zu halten, um möglichst wenig Fragen offen zu lassen. Sollten jedoch trotzdem Fragen oder Unklarheiten auftreten, kann man mich unter e8971103@student.tuwien.ac.at erreichen.

Matthias Labschütz

0. How To Use:

0.1 Config:

Datei 'config.txt' im Quake III Arena Ordner.

mouse_locked	Startwert für Maus, 0 = default Shooter Ansicht, 1 = Maus bewegt sich über den Bildschirm
reset_after_hit	0 = nein, 1 = ja, bei einem Treffer wird die Maus in die Mitte des Bildschirmes zurückgesetzt und eine gewisse Zeit eingefroren
use_mosquito_list	0 = nein, 1 = ja, verwendet eine Liste der Mückenpositionen die der Reiche nach durchgegangen wird. Am Ende der Liste wird wieder der erste Eintrag gewählt.
mosquito_list	spezifiziert eine Liste der Mückenpositionen die der Reihe nach abgearbeitet wird, zulässig sind Werte von 2-9. z.B: 2 3 6 8 9
hrtf_sound	0 = nein, 1 = ja, aktiviert räumlichen Sound, wenn deaktiviert wird der Sound nicht räumlich versetzt
frag_limit	definiert die Anzahl der Mücken die man erschlagen muss

0.2 EXE/Verknüpfung:

Einfach die Verknüpfung starten.

+set sv_pure … sollte immer aktiviert sein

0.3 Aufzeichnung:

Wird in 'mouse_pos.txt' gespeichert. Siehe 1.3.3.

1. Dokumentation:

1.1 Engine:

Zur Umsetzung des Projektes wurde die ioquake 3 Engine Version 2008-04-04 (http://ioquake3.org/), eine Modifikation der Quake 3 (id tech 3) Engine, die auf der id Seite zum Download angeboten wird. Die Engine ist in C geschrieben.

1.1.1 Die Projektstruktur:

Der Projekt Ordner teilt sich in 4 Ordner:

Build: beinhaltet die Ordner der builds, jeder Kompiliervorgang erzeugt bis zu 4 verschiedene build Dateien (ioquake3.exe, cgamex86.dll, qagamex86.dll, uix86.dll). Diese müssen nach dem kompilieren in den Quake III Arena Ordner kopiert werden. Die *.exe Datei direkt in diesen Ordener. Die *.dll Files werden in den baseq3 Ordner kopiert.

Code: beinhaltet den Quellcode.

Misc: im msvc Ordner befinden sich die Projekt Dateien. Das Projekt ioq3 bezeichnet das verwendete Projekt. Cgame, game, q3_ui, quake3 und ui sind untergeordnete Projekte die sich auch im ioq3 Projekt finden.

Ui: unwichtig

1.1.2 Die Engine:

Wichtig ist hier nur der Engine Ordner (Quake III Arena) und der baseq3 Ordner.

Im **Engine Ordner** sind die 2 Ausführbaren:

ioquake3.exe und Mosquito_Experiment (Mosquito_Experiment aktiviert den sv_pure Modus der notwendig ist um eigenen Quellcode, bzw. speziell modifizierten Quellcode zu verwenden und startet die richtige Map.)

Des weiteren wird in 'config.txt' die Experiment spezifische Konfiguration gesetzt. Und in 'mouse_pos.txt' die Mauspositionen gespeichert.

Im **baseq3 Ordner** befinden sich unter anderem die q3config.cfg (zum setzen der Quake eigenen Config Datei) und die pak0.pk3 – pak8.pk3 Archive (die man mit z.b. winrar öffnen kann) die den Content beinhalten.

Der **Gamecontent** setzt sich zusammen aus (jeweils nur die wichtigen, bzw. verwendeten):

maps: beinhaltet die Levels. (*.map Editor File und *.bsp kompiliertes File, *.aas ist unwichtig da nur für Bots benötigt) Eigene Levels: mosquito, mosquito_1, mosquito_2

models: beinhaltet die Modelle. Sowohl Spieler Modelle (die nicht geändert wurden). Als auch Modelle die innerhalb der Maps als mapmodels verwendet wurden. Alle eigenen Modelle und Texturen befinden sich im /gelse Unterordner.

scripts: hier befinden sich *.shader Dateien die für spezielle Oberflächeneigenschaften der Modelle und Texturen zuständig sind. Sollten eigene *.shader Dateien geschrieben werden sollten diese in der 'shaderlist.txt' eingetragen werden. Eigens erstellt wurde die 'gelse.shader' Datei.

sound: für die Sounds. Selbst erstellt wurde der '/gelse' Unterordner.

textures: für Texturen innerhalb der Levels. Wieder gibt es hier einen '/gelse' Unterordner.

1.1.3 Der Leveleditor:

Als Editor zum Erstellen der Maps wurde der GtkRadiant 1.5 verwendet (http://www.qeradiant.com/cgi-bin/trac.cgi). Hier ist der q3.game Ordner wichtig da er den Teil für Quake 3 zu Verfügung stellt. Im speziellen die Datei: 'q3.game/baseq3/entities.ent'. Diese liefert die Entity Definitionen für eigens erstellte Entities die benötigt wurden. (Diese sind im File mit M: kommentiert.)

Anm.: Entities sind interaktive Elemente innerhalb einer Map (z.B. Spielerstart, eine Türe, ein Waffenspawn, ...), die gesondert im Editor platziert und eingestellt werden können. (Zu Entities gibt es zur konkreten Behandlung zugehörige Codestellen in Sourcecode pro Entity.)

1.2 Allgemeines

Änderungen die von mir an der ioquake Engine gemacht wurden, wurden mit '//**M:**' kommentiert. Somit sind sie beim durchsuchen des Quellcodes leichter auffindbar.

Die Engine ist wie oben genannt in unterschiedliche Module aufgeteilt. Diese sind cgame, game, quake3 und q3_ui. (q3_ui ist für die Umsetzung nicht so wichtig, ui wird nicht verwendet) Meine Änderungen befinden sich in all diesen Projekten. Die Module arbeiten über diverse 'VM_Calls' zusammen. Genaueres im Einzelfall ergibt sich oft beim durchsuchen des Quellcodes, jedoch ist die Kommunikation der Module untereinander oft etwas schwerer zu rekonstruieren.

Jedes der 3 wichtigen Module (cgame, game, quake3) hat eine main.c.

Die Konsole öffnet man mit SHIFT+ESC.

CD-Key könnte notwendig sein. Der Content der Quake Engine wurde nicht zum Download freigegeben.

1.3 Die Änderungen im Detail:

1.3.1 Maus einfrieren:

Eine zentrale Anforderung war es die Engine so zu modifizieren, dass es möglich ist die Maus in einer fixen Ansicht getrennt zu bewegen. Sprich die Maus bewegt sich über den Bildschirm und nicht die Ansicht ändert sich bei Mausbewegung.

Die Umsetzung funktioniert prinzipiell wie folgt:

In der config.txt kann man den Startwert der Maus (locked oder nicht) mit 'mouse_locked 0' setzen.

Beim Drücken der Taste 'm' im Spiel kann man zwischen den Mausansichten wechseln.

Der Befehl 'mouselock' kann zum Beispiel in der Quake 3 config an eine Taste gebunden werden. (bei mir m).

Umsetzung im Code:

Das eigentliche Locken passiert in 'cl_input.c' in 'CL_MouseMove'.

'IN_MouseLock' wird aufgerufen wenn der Befehl 'mouselock' ausgeführt wird.Als z.B. Beim drücken der entsprechenden Taste.
Praktischer Weise werden hier auch dementsprechend die viewangles modifiziert, sodass nur hier

relativ lokal im Code etwas geändert werden musste.

1.3.2 Maus getrennt bewegen:

Umsetzung im Code:

Passieren im quake3 und cgame Modul. In quake3 ist der Input, in cgame wird gezeichnet.

Die Mauspositionen die in 'cl_input.c' geändert werden, werden an 'cl_main.c' zurückgeliefert und von dort aus auch an das cgame Modul geschickt.

Dort wird in der Datei 'cg_draw.c' die Funktion 'CG_DrawCrosshair' sohingehend abgewandelt, dass das Crosshair an einer anderen Position als der Mitte gezeichnet wird! (x, y Offset)

1.3.3 Maus Positionen speichern:

Um das Experiment aufzuzeichnen werden die Mauspositionen und deren Zeiten, sowie Trefferzeitpunkte einer Mücke gespeichert. Das Speichern passiert immer in der derzeitigen Umsetzung. Dazu wird immer in die Datei 'mouse_pos.txt' gespeichert. Wenn keine vorhanden wird sie erstellt.

In der Datei wird gespeichert:

'Zeit in ms/Mausposition X/ Mausposition Y'

Beim Treffen einer Mücke wird zusätzlich die Zeile 'last was hit' gespeichert. Vorherige Position ist die Mausposition und der Trefferzeitpunkt.

Beim Treffen der letzten Mücke wird zusätzlich die Zeile 'game over' gespeichert und das Speichern wird beendet.

Umsetzung im Code:

Hier ist die Datei 'cl_main.c' zentral. Die Funktion 'CL_saveMousePosition' wird pro Frame ausgeführt. Hier wird entweder in ein Array mit bestimmter Länge gespeichert, oder das Array ins File übertragen.

1.3.4 Eigene Direct Sound Implementation innerhalb der Engine:

Eine andere Anforderung war es hrtf Sound zu verwenden. Dazu habe ich mich für directsound entschlossen.

Die Umsetzung funktioniert prinzipiell wie folgt:

In Visual Studio ist die Preprocessor Definition 'USE_DSOUND' einzustellen um den Sound zu verwenden. (Bereits eingestellt. Alternativ gibt es auch 'USE_OPENAL' aus ioquake.)

Im quake3 Modul wurde der Ordner '/sound' hinzugefügt, der sich um die Implementation von directsound kümmert. Leider konnte ich diesen nicht objektorientiert genug implementieren, da das in c nur über Umwege möglich ist und der Sound hier recht Quake spezifisch behandelt wird. Zugehöriges cpp Projekt ist jedoch auch vorhanden. Die Änderung für die Portierung nach c waren jedoch zu groß um kompatibel zu bleiben. (Derzeit müsste man sich händisch darum kümmern c und cpp gleichtzeitig zu ändern.)

Umsetzung im Code:

Die zentral wichtige Position im quake3 Modul ist 'snd_main.c'. Von dort werden die jeweiligen

Funktionen aus der eigenen Implementation in '/sound' aufgerufen.

Der Großteil ist recht selbsterklärend, 3 Punkte sind jedoch vielleicht wichtig.

Quake benötigt Sounds die an Entities angehängt sind. Also jeder Sound hängt quasi an einem Entity. Die Verbindung Entity-Soundquelle musste deshalb für Quake extra hergestellt werden. (Das ist für eine allgemeine Sound Implementation nicht notwendig.) Dazu werden in ein Array die Indizes geschrieben.

Die 'S_Update' wird immer wieder aufgerufen um den Sound upzudaten. Es gibt auch die Option jegliche Änderungen (z.B. Positionswechsel der Soundquellen) sofort auszuführen. Jedoch ist es Störungsfreier wenn man erst alle Änderungen ausführt und dann einmal (z.B. pro Frame) ein Sound Update ausführt.

Das Koordinatensystem von Quake und das von directsound mussten zusammengehängt werden. DirectSound benötigt einen front und einen up-Vektor. Hier war eine Transformation in das jeweils andere Koordinatensystem notwendig um richtige Resultate zu erziehlen.

1.3.5 Config File:

Dazu wurde der Ordner '/config' im quake3 und game Modul eingebunden.

Umsetzung im Code:

'config.h', 'config.c'

Die Umsetzung ist in den 2 Dateien ist recht selbsterklärend. Die Parameter der Config werden von der Engine zwecks Einfachheit im Modul quake3 und game eingelesen und verwendet. Andernfalls hätte man die Parameter mittels Kommunikation zwischen den Modulen verschicken müssen. Dh wird die config 2 mal geladen.

In Visual Studio ist die Preprocessor Definition 'USE_CONFIG' für beiden Module einzustellen um die Config zu verwenden.

1.3.6 Die Spiellogik:

Die Spiellogik war etwas aufwendiger umzusetzen. Dazu mussten eigene Entities im Level Editor erstellt werden und diese im Editor platziert und in der Engine programmiert werden.

Die Umsetzung funktioniert prinzipiell wie folgt:

Der Spieler befindet sich im Level an einer Startposition. Diese ist gleichzeitig ein Auslöser (Trigger) für die Spiellogik.

Es exisiteren eine schwarze Wand vor dem Spieler, die das “ausgeschaltete Licht” simulieren soll.Bei der Wand handelt es sich um eine Türe (func_door).

Mücken die ebenfalls eine modifiziertes func_door sind. Ein func_door ist ein generell bewegliches Entity, dass 2 Zustände annehmen kann, offen und geschlossen. Die Mücke kann 2 Zustände annehmen lebendig an der Wand, oder tot hinter der Wand. Ist die Mücke lebendig kann sie bei einem Treffer (Zufügen von Schaden) tot 'geschossen' werden und bewegt sich hinter die Wand. Ist die Mücke tot kann sie quasi wiederbelebt werden wenn sie von einem anderen Entity aktiviert wird.

Ein “Verteiler” Entity kümmert sich um das aktivieren von Mücken (per Zufall oder per Liste). Und um das aktivieren der schwarzen Wand.

Umsetzung im Editor:

Hier nochmal die Referenz auf 'entities.ent' das benötigt wird um Entities für den Editor zu definieren. (siehe auch 1.1.3)

Das Verteiler Entity nennt sich 'target_delay_twice_random'. Dieses braucht als Parameter: target: die schwarze Wand; target1-target9: die Mücken (alle Targets sollten gesetzt werden)

Die Mücken sind 'func_door_gelse'. Sie sollten 'is_switch' und 'start_open' aktiviert haben.

Die schwarze Wand ist ein einfaches 'func_door' ohne Bewegungssound und der Paramter wait gibt an wie lange es dunkel bleibt.

Um das System zu starten muss eine Mücke die am Verteilerentity hängt aktiviert werden. Sobald diese “tot geschossen” wird wird der Verteiler zum ersten mal aktiviert und sucht sich eine neue Mücke und aktiviert die Wand.

Umsetzung im Code:

Entities werden generell im game Modul programmiert.

Das Verteiler Entity befindet sich in 'g_target.c' als 'SP_target_delay_twice_random'. Das Mücken Entity in 'g_mover.c' als 'SP_func_door_gelse'. Zur konkreten Implementation gibt es nicht viel zu sagen. Das einbinden und weiterleiten der Funktionen im game Modul war etwas aufwendiger ist jedoch mit Suchfunktion nachvollziehbar.

Es werden hier config Parameter verwendet um zu entscheiden ob das Verteiler Entity eine Liste, oder Zufallswerte verwenden soll.

Wichtig für den Zufallsmodus ist, dass man alle 9 Targets setzt. Beim Listenmodus sollte man eine Liste in der config definieren.

Das default Entity wurde um die Targets 1-9 erweitert.

Sounds wurden deaktiviert und es wurde ermöglicht eine Türe als switch zu benutzen die sich nicht automatisch wieder nach einer bestimmten Zeit in die Ausgangsposition zurückbewegt. Also wurde auch das default func_door als switch door überschrieben.

1.3.7 Kleine Änderungen am Spiel:

- *Waffe Fliegenklatsche:* dazu wurde einfach nur die default Maschinengewehr Waffe abgeändert. Der Sound wurde ersetzt, dieGeschwindigkeit und der Schaden reduziert und die Modelle und Treffer-Sprites vorerst auskommentiert da Einschusslöcher und eine Waffe mit Fliegenklatschen Sound nicht so realistisch wirken. Änderungen in: 'bg_misc.c', 'bg_pmove.c', 'cg_main.c', 'cg_weapons.c'
- *Teleporter Effekt bei Spawn:* auskommentiert in 'cg_effects.c'
- *reset after hit:* setzt die Maus wieder in Ausgangsposition zurück. In 'cl_input.c'.

1.4 Levleditor hilfreiches/Problemfälle

- *Custom Modelle:* haben bei mir nur im *.ase Format funktioniert. Dazu exportiert man das Modell als *.ase (beispielweise in Maya mit ActiveX). Dann muss man im File die Pfade zu den Modelltexturen setzen. MATERIAL_NAME und BITMAP müssen den Texturpfad setzen.

(z.B. "models\gelse\book_test.tga") Das Modell kann man im Editor als *.ase (misc_model) laden.

- *Transparente/halbtransparente Texturen* brauchen einen eigens geschriebenen Shader.
- *Hilfreiche Seite:* http://www.killerlevels.de/main.php?page=tutorials.include

2. Bericht

Warum die Quake Engine?

Weil sie die neueste Kommerzielle Open Source Engine ist. Möglicherweise wären andere Engines z.B. Ogre für ein Experiment sinnvoller, da sie meiner Vermutung nach im konkreten Fall flexibler einzusetzen sind.

Vorteile der Quake Engine sind jedoch Unterstützung von Multiplayer. Multiplayer Experimente wären theoretisch möglich. Es gibt enorm viele Spiele die auf der Quake Engine aufgebaut waren. (Liste findet sich leicht auf Wikipedia.) Also kommerzieller Erfolg, viel Content (der jedoch teilweise schon veraltet ist und nicht so frei zugänglich). Hoffentlich relativ problemlose Anwendung auf anderen Computern. Weiters war die Engine meiner Vermutung nach in einigen Bereichen ein maßgeblich für die Art der Weiterentwicklung jetziger Hard und Software im Grafikbereich.

Nachteile sind wenig Flexibilität im Aufbau. Was sich zum Beispiel im eigenen Leveleditor zeigt, der zwar eine Trennung zwischen Content und Engine ermöglicht, also einfaches Erstellen von Levels für Nutzer die keinen Zugang zum Quellcode haben. Dadurch entsteht aber auch Mehraufwand für den Entwickler. Auch der Aufbau in Module ist gerade für Experimente ohne Multiplayer nur zusätzlicher Mehraufwand.

Ein weiterer großer Nachteil ist fehlende Dokumentation der Engine. Änderungen muss man sich großteils selbst ableiten aus dem was vorhanden ist. (Ich denke eine Dokumentation bzw Übersicht über die Engine zu schreiben wäre eine sehr nützliche Sache, da man dazu kaum etwas im Internet findet.) Eine online Plattform, Wiki oder ähnliches habe ich nicht gefunden.

Warum DirectSound? Warum wurde der Sound umgesetzt wie er umgesetzt wurde?

Anforderung war HRTF Rendering. Es gibt derzeit keine Programmbibliothek die individuelle HRTF unterstützt. (Das wird sich vermutlich auch nicht ändern.) Eine nicht individuelle HRTF wird in z.B. In DirectSound, Fmod oder OpenAl mit spezieller Sounkarte unterstützt. Im Fmod Forum stand, dass HRTF Sound nicht korrekt erzeugt wird sondern nur angenähert. Bei OpenAl Bräuchte man eine spezielle Soundkarte. Bei DirectSound gibt es keine näheren Angaben wie das HRTF Verfahren genau umgesetzt wird. Aber auch nicht, dass es unkorrekt ist. Jedenfalls sind unterschiede im Klang hörbar je nach Position der Soundquelle.

Die Umsetzung ist leider nicht so schön geworden wie geplant. Da die Engine in C geschrieben ist und objektorientiertes Programmieren in C nicht direkt unterstützt wird und erst simuliert werden müsste. Erste Herangehensweise war es gewesen die Bibilithek in C++ einzubinden und dieses im C-Code zu verwenden. Es hat sich aber herrausgestellt, dass DirectSound in C andere Befehle verwendet als in C++. Jedenfalls bin ich dann dazu übergegangen alles auf C umzuschreiben. Der Aufbau bleib relativ gleich, jedoch müsste man immer wenn man etwas im C-Teil schreibt den C++ Teil dementsprechend abändern. Weiters braucht die Engine gewisse Parameter die man vermutlich beim Verwenden von 3D Sound in anderen Engines/Anwendungen nicht brauchen würde. Somit ist die Implementation recht Quake spezifisch in C. Für Vererbung ect. wäre es vermutlich in C++ schöner. (Die Implementation in

C++ habe ich als gesondertes Projekt verfügbar und kann sie auf Anfrage auch weitergeben.)

Kurze Reihenfolge der Implemetation:

Bis Weihnachten 09:

- einfaches Level
- Umsetzung der Spiellogik
- Fadenkreuz fix/beweglich
- einfaches Speichern der Mausbewegung

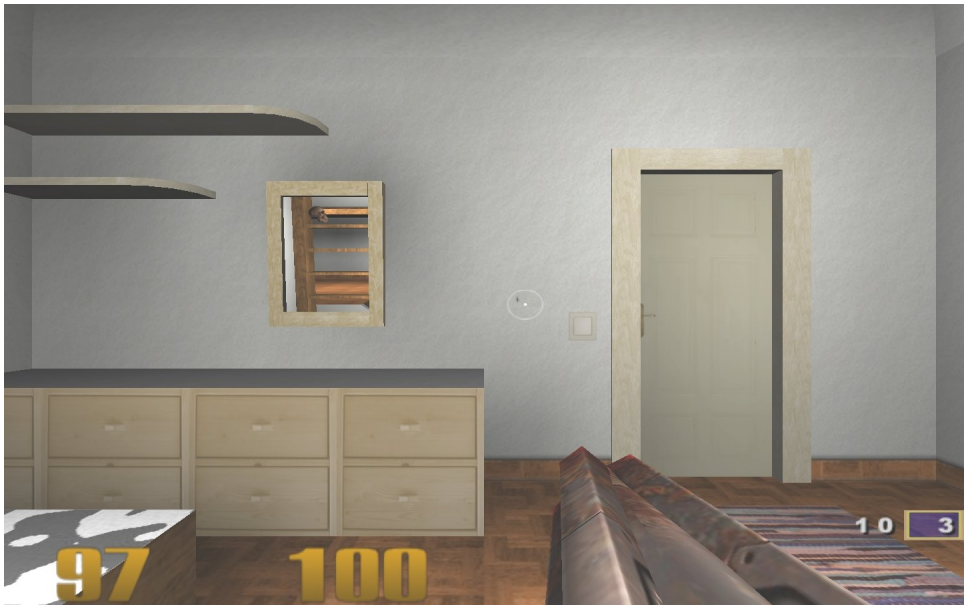

Abbildung 1: Screenshot Weihnachten 09

Bis Februar 10:

- Details ausbessern (Waffe, Spawn, ...)
- Umsetzung der Soundengine
- Config
- Level grafisch verschönern bzw unübersichtlicher machen
- Erweitertes Speichern

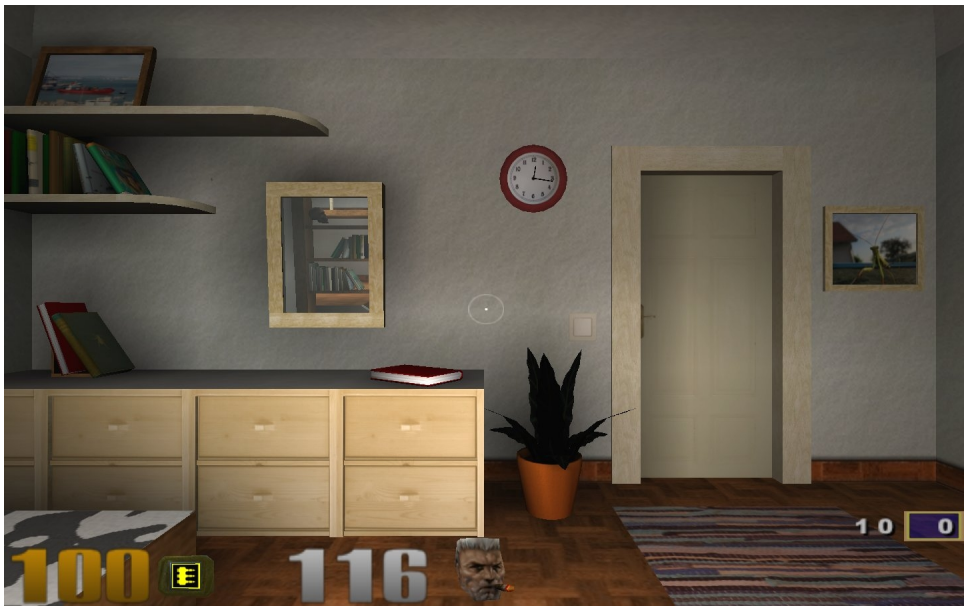

Abbildung 2: Screenshot Februar 10

3. Experiment Protokoll Vorschlag

3.1 Experiment: Bimodal Task-Facilitation in a Virtual Localisation and Pointing Simulation through Spatialized Sound Rendering

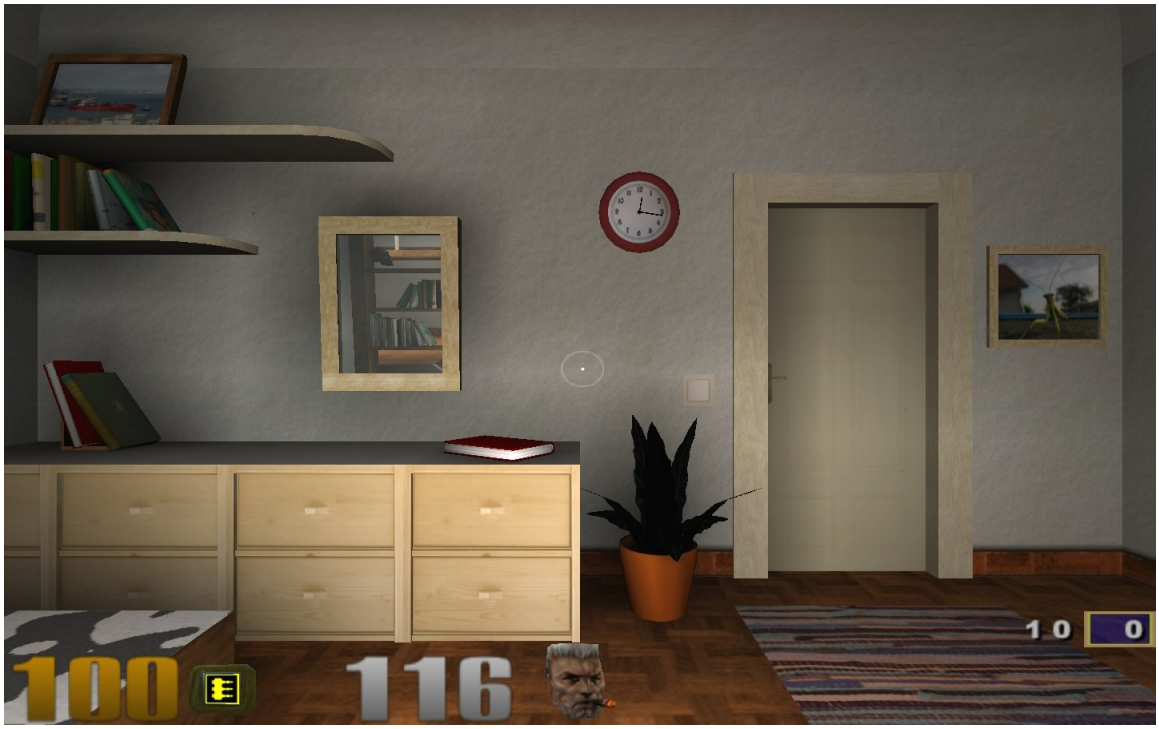

Figure 1: Screenshot of the virtual experiment scene

Matthias Bernhard

Matthias Labschütz

Date:

Recorders:

Name	Signature

3.2 Introduction

The task of localizing and clicking objects is part of almost every virtual environment simulation. The probably most immersive way of interacting with todays virtual environments is playing shooter games. Players need sounds to localize other players that are not visible. For example obstructed by other objects or behind the player. It seems obvious that spatial sound rendering as already used in games is an essential part of increasing the player's performance.

However, this experiment investigates the influence of spatial sound, in the task of finding and clicking already visible objects.

3.3 Hypothesis

This experiment tries to show, if the task of localizing and clicking a barely visible object is improved using spatial sound, causing bimodal task-facilitation.

The participants are using a mouse to click an object emiting sound in a virtual environment with and without spatial sound. Times to hit the target will be saved and compared.

3.4 Results

To come.

3.5 Method

The experiment uses either a video beamer or a standard monitor (to be specified). Participants are using headphones. They will be playing 3 games of hitting mosquitos.

Every participant starts off with a test game making sure they understood the game mechanics. The test game uses spatialized sound.

There are 2 groups of participants. Each group plays two additional games, one game using spatialized and one game using non-spatialized sound. Group A will start with non-spatialized sound. Group B will start with spatialized sound.

There are 8 predifined locations of sound sources.

A list of locations of sound sources is specified. The first game of all participants always uses the same list of locations, the second game of all participants uses a list differing from the first game. The number of sound sources in the list is the same in both games, but the order at which they are executed is different. Patterns of locations in a row should not be used to minimize memorizing effects of patterns.

The time to find a certain sound source (e.g sound source location 3), for each participant, is compared with his/her times to locate the same sound source at different tries. This way the times to locate a sound source for a certain player can be compared using spatial and non-spatial results. Times between participants are not planned to be compared, since they can differ by the skill factor of a participant in using a mouse or clicking things.

The participants are not being told they will hear different sounds (spatial/non-spatial). If they ask, they get the answer that this is part of the experiment and the sound is indeed different.

Partitioning the participants in 2 groups will allow to compare their results for experiment errors.

3.5 Discussion

Based on results.