3D GameStudio

Raumflug-Workshop

von Nicholas Chionilos / February 2001

Inhalt

Willkommen beim Höhenflug mit diesem Raumflug-Workshop! Diesen Kurs habe ich als Hilfestellung für Leute konzipiert, die ihr eigenes 3D Raum-Level mit 3D GameStudio entwickeln wollen. Hierbei verwende ich einige Features, die erst mit dem letzten Update (4.22) verfügbar sind.

Wie andere zuvor, zielt auch der vorliegende Kurs vor allem auf Anwender, die bereits einige Vorerfahrungen im Umgang mit 3D GameStudio mitbringen. Ich setze voraus, dass Sie das Tutorial durchgearbeitet haben und mit dem Handwerkszeug (WED, MED, und WDL) umzugehen wissen.

Das **spaceshp.wdl**-Skript, das diesem Workshop beigefügt ist, wurde als Simulation eines Raumschiffes kreiert, welches sich in einem schwerelosen Umfeld bewegt. Es beinhaltet drei verschiedene Kamerasichten, vier Kontoll-Flags und fünf Parameter, die eine individuelle Anpassung der Flugeigenschaften Ihres Schiffs erlauben.

Dieser Text ist dazu gedacht, die mitgelieferte Dokumentation von 3D GameStudio zu ergänzen, nicht zu ersetzen. Falls Ihnen irgendwas in diesem Workshop unklar ist, lesen Sie bitte im Handbuch von 3D GameStudio nach.

Schliesslich möchte ich mich noch besonders bedanken bei:

- Remi, weil er mich dem Konzept der Himmelskugel vertraut machte
- Kyodai, für seine grosszügige Spende des Schiffs-Models, das hier verwendet wird.
- Vivi, für die sehr coole Schiffshaut (Skin)
- JCL, der das Chase Cam-Skript beisteuerte
- Doug, für seine unendliche Geduld und Beratung in allen Belangen

Ich hoffe, Sie haben Spass an diesem Worksphop! Fragen und Anmerkungen können an **Nicholas.Chionilos@Vertexconsulting.com** gesendet werden und ich will mein Bestes tun, sie zu beantworten.

-Nick "WildCat" Chionilos.

Bevor wir anfangen:

Besorgen Sie sich die neueste Version

Bervor Sie loslegen, sollten Sie sicher sein, dass Sie mit der neuesten Version von 3D GameStuio (4.22 oder neuer) arbeiten. Wir werden nämlich einige der neu hinzugekommenen Features verwenden.

Bereiten Sie Ihre Arbeitsumgebung vor

Erstellen Sie in Ihrem Gstudio-Ordner eine neue Datei mit dem Namen **"Space"**. Das ist nun der Ordner, in dem Sie sämtliche Elemente Ihres Spieles ablegen.

Entzippen Sie den Inhalt von **`space.zip`** und speichern Sie die Dateien in diesen neuen Ordner. Nun sollten folgende Dateien im "**Space**"- Verzeichnis enthalten sein:

```
spaceshp.wdl
spaceshp.mdl
sphere.mdl
starmap.pcx
stars.mdl
engine.wav
thrustr.wav
```

Fangen wir an!

Einen Level im Universum anzusiedeln ist schon ein wenig anders, als ein Baller- oder Rollenspiel oder sogar einen Flugsimulator zu erstellen. Der grösste Unterschied besteht darin, dass bei den meisten konventionellen Leveln eine stationäre sogenannte "Sky Box" verwendet wird. Diese unverrückbare "Himmelskiste" wird per WED definiert und umschliesst die gesamte Welt. Ein Universums-Level braucht hingegen eine "Sternenkugel" ("Star Sphere"): ein im MED entwickeltes, bewegliches Model, das das Level umschliesst und sich mit dem Raumschiff mitbewegt.

In folgenden Schritten werden wir eine All-Umgebung erschaffen:

1. Kreieren einer Sternenkugel, womit wir festlegen, wie Ihr Universum aussieht
2. Erstellen eines gossen Levels in WED
3. Plazieren eines Schiffs im Zentrum der Welt
4. Zuweisen der **`player_ship-`**Aktion

Kreiern einer Sternenkugel

Die Sternenkugel ist einfach ein sehr grosses Kugel-Model, welches den Ausblick ins All mit seinen Sternen repräsentiert, den wir aus unserem Raumschiff haben. Wir erschaffen dieses Universum in vier Stufen:

1. Erstellen eines Kugel-Models
2. Umklappen der Seiten des Polygons, um damit de facto die Innenseiten nach aussen zu kehren
3. Skalieren der Kugel zu einer passenden Grösse.
4. Das Model mit einer All-Oberfläche, der Skin, überziehen

Zwar ist die Theorie zur Erstellung einer Sternenkugel sehr simpel, soll die Sternen-Map aber professionell aussehen, sind ein paar praktische Überlegungen miteinzubeziehen:

- Je mehr Polygone in Ihrer Kugel benutzt werden, je geringer wird die Verzerrung, die Sie an den Kanten Ihres Sichtfeldes zu sehen kriegen, wenn Sie sich im Spiel drehen. Ich empfehle zwischen 500 und 1200 Polygonen.

- Je grösser Sie Ihre Kugel machen, desto weiter sind die Polygone, aus denen sie geformt ist, von der Kamera weg. Mit zunehmender Grösse wird es also immer unwahrscheinlicher, die Konturen einzelner Polygone zu erkennen.

- Je einheitlicher Sie die einzelnen Polygone gestalten, je geringer ist die Wahrscheinlichkeit von Unregelmässigkeiten im Sternenrund.

MED ist nicht unbedingt die beste Ausgangsbasis um eine Kugel neu zu erstellen. Das Basisobjekt (Primitive) hat zu wenig Polygone und wenn Sie deren Anzahl mit dem Unterteilungswerkzeug erhöhen, erhalten Sie Polygone von unterschiedlicher Form. Aus diesen Gründen empfehle ich, dass Sie entweder ein Polygon aus einem anderen Model-Editor, wie Milkshape importieren, oder verwenden Sie das **`sphere.mdl`**, das ich Ihnen als Grundausstattung bereits mitgegeben habe.

Genug geredet. Öffnen Sie Ihren MED-Editor und lassen Sie uns den Himmel erschaffen!

Stufe 1: Am Anfang ist die Kugel im MED.

Wir fangen an, indem wir das hier mitgelieferte **sphere.mdl** laden. Dieses Model hat 720 Polygone, die allesamt eine einheitliche Form haben und ist somit eine exzellente Ausgangsbasis für unsere Sternenkugel.

Öffnen Sie MED und gehen auf `file→open` dann wählen Sie **sphere.mdl** aus Ihrem "**Space**"-Ordner.

Ihr Bildschirm sollte nun in etwa so aussehen:

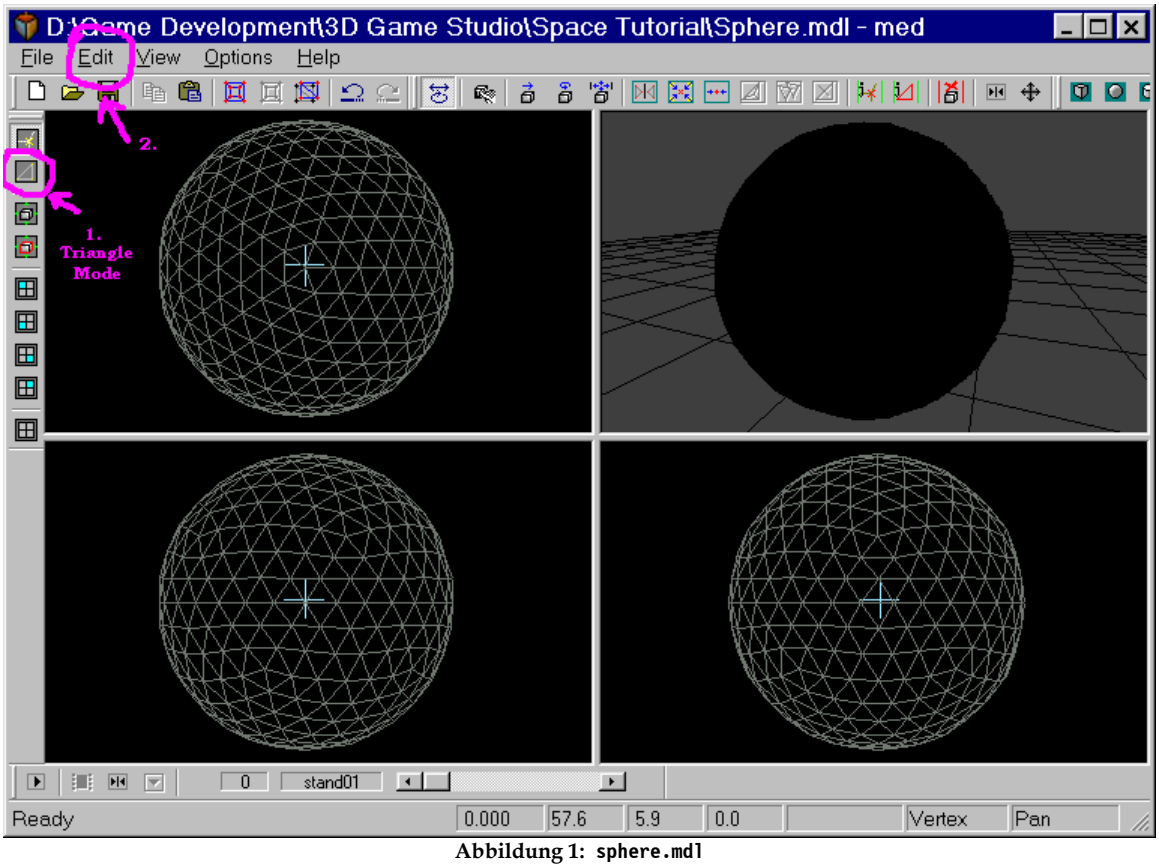

Abbildung 1: sphere.mdl

Bevor wir weiter machen, wollen wir sicher sein, dass wir auch im Dreiecks-Modus (Triangle Mode) arbeiten. Klicken Sie also auf die entprechende Schaltfläche in der linken Knopfleiste. Ich habe sie in Abb. 1 mit einem rosa Kringel und der Nummer 1 gekennzeichnet.

Gehen Sie ins Editier-Menü und auf `select all` (alles auswählen)
Nun sollten Sie sämtliche Polygone der Kugel goldleuchtend sehen; genauso, wie in Abbildung 2:

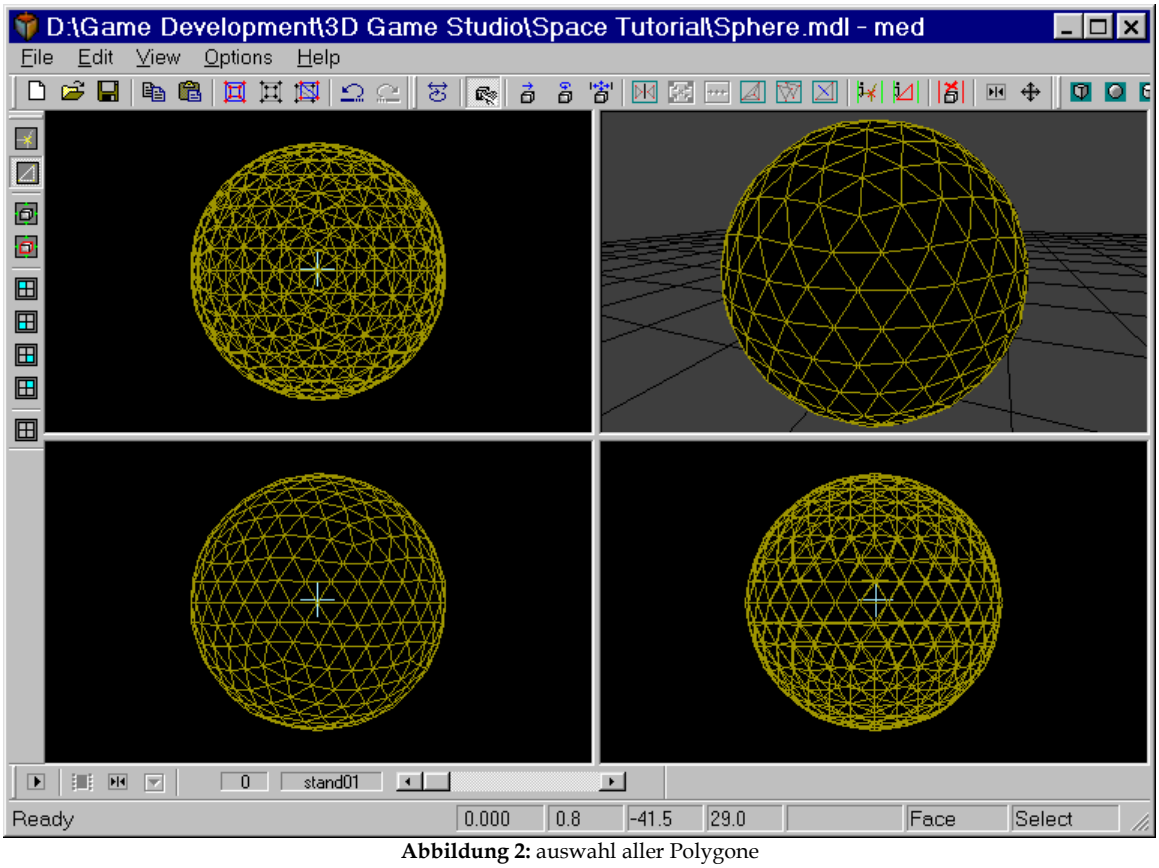

Abbildung 2: auswahl aller Polygone

Step 2: Kippen der Normalen

Eine Normale ist schlicht die Richtung, in die die sichtbare Seite eines Polygons zeigt. Betrachten Sie die Normalen, indem Sie auf View `options` → `show normals` → `show all` gehen. Das Resultat sollte so aussehen, wie in Abbildung 3.

Sie stellen fest, dass das Model im Moment "stachelig" aussieht. Das kommt daher, dass jedes Polygon eine kleine Linie aufweist, die aus ihrem Zentrum kommt und von dort geradewegs vom Zentrum des gesamten Models weg zeigt. Wir sehen also, dass alle Polygone zunächst nach aussen gerichtet sind.

Weshalb müssen wir so genau darauf achten, wohin die Normalen zeigen? Der Grund ist der, dass die Oberfläche (Skin), die wir dieser Kugel aufspannen, nur von der Richtung aus zu sehen ist, in die diese Normalen weisen. Wenn also die Normalen nach aussen zeigen, können Sie die Oberfläche des betreffenden Objektes nur dann sehen, wenn Sie von aussen drauf schauen. Das wäre ganz schön, wenn wir einen Planeten zum anschauen haben wollten.

Da sich unser Schiff aber **innerhalb** der Kugel auf die Reise machen wird, müssen wir die Normalen umklappen, so dass sie ins Innere, zum Zentrum der Kugel hin, zeigen. Auf diese Weise können wir die Sternenoberfläche dann sehen, wenn sich unsere Kamera innerhalb des Models befindet. Tun Sie also, was nötig ist und klicken Sie auf den `flip normals`-Knopf in der Werkzeugleiste (s. Abb. 3).

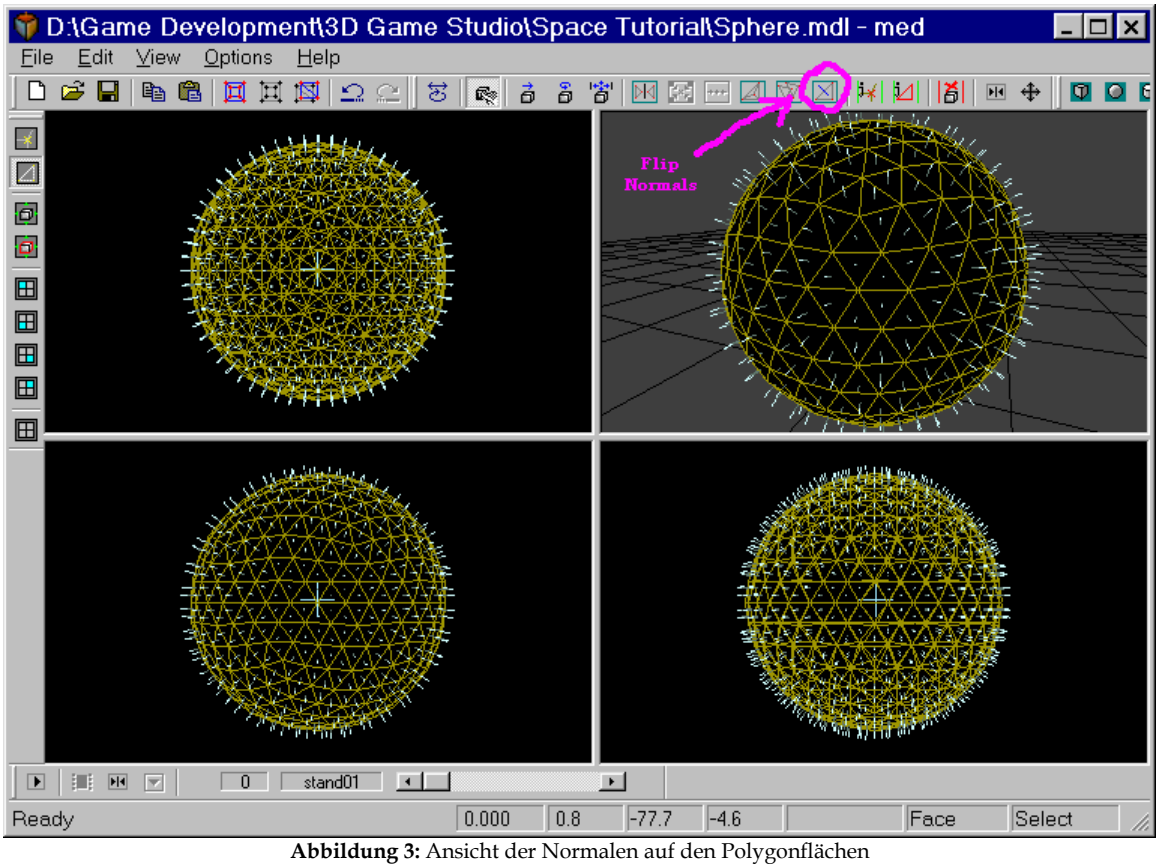

Abbildung 3: Ansicht der Normalen auf den Polygonflächen

Nun sollten Sie, wie in Abb. 4, ein Kugel-Model haben, dessen Normale alle nach innen zeigen.

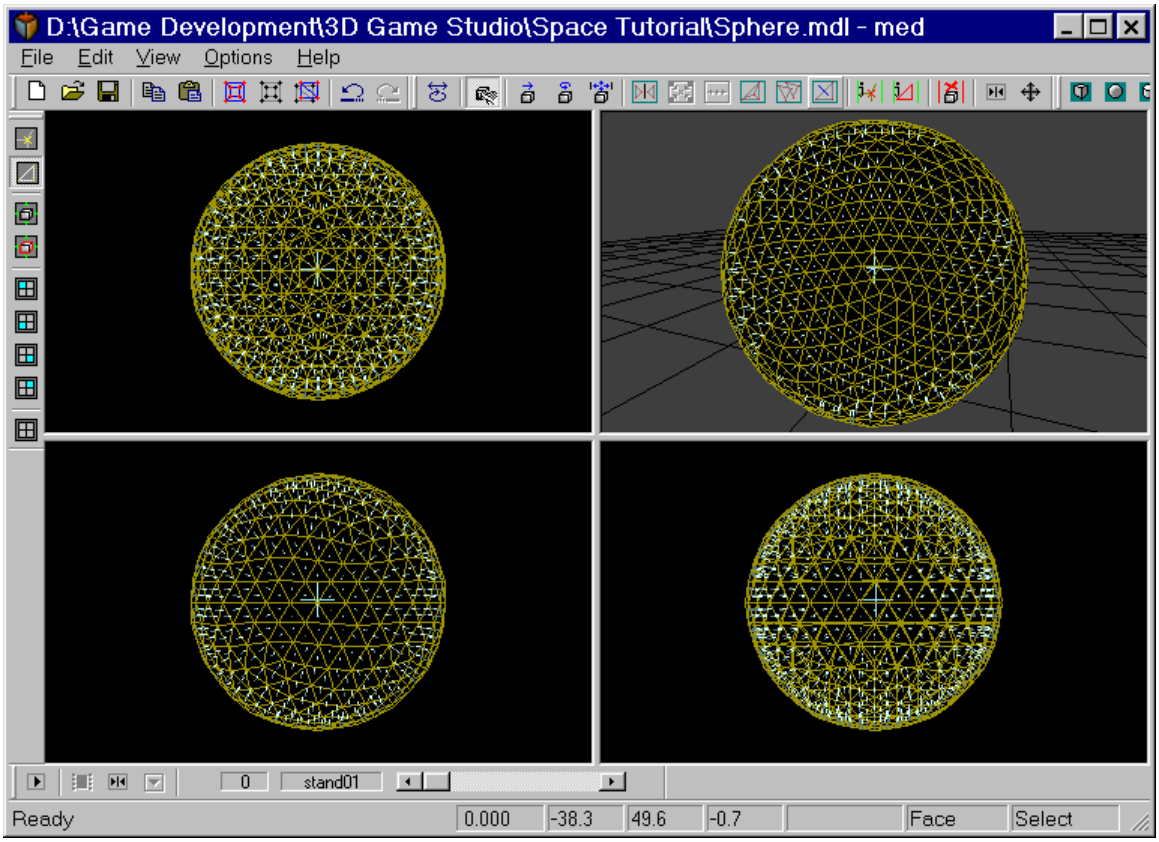

Abbildung 4: Kugel mit nach innen "geflippten" Normalen

Sie können nun mit `options` → `show normals` → `none` zur normalen Model-Ansicht zurückkehren.

Step 3: Skalieren des Models

Skalieren Sie das Model auf eine ordenliche Grösse. Ich schlage etwa 12.000 Einheiten entlang jeder Achse vor. Machen Sie das, indem Sie, wie in Abbildung 5 gezeigt, das Positions-Werkzeug aus der Knopfleiste auswählen. Klicken Sie nun rechts, halten die Maustaste gedrückt und ziehen Sie den Cursor nach unten.

Diese Operation hat den Effekt, dass die Kamera weit von Ihrer Kugel weggezogen wird und diese dadurch ziemlich klein erscheint. Testen Sie die Distanz indem Sie Ihren Cursor dicht an die rechte Grenze der Draufsicht (Top View) bringen und sich vergewissern, dass, wie in Abbildung 5, die Y-Verschiebung ausreichend gross – etwa 12.000 – ist.

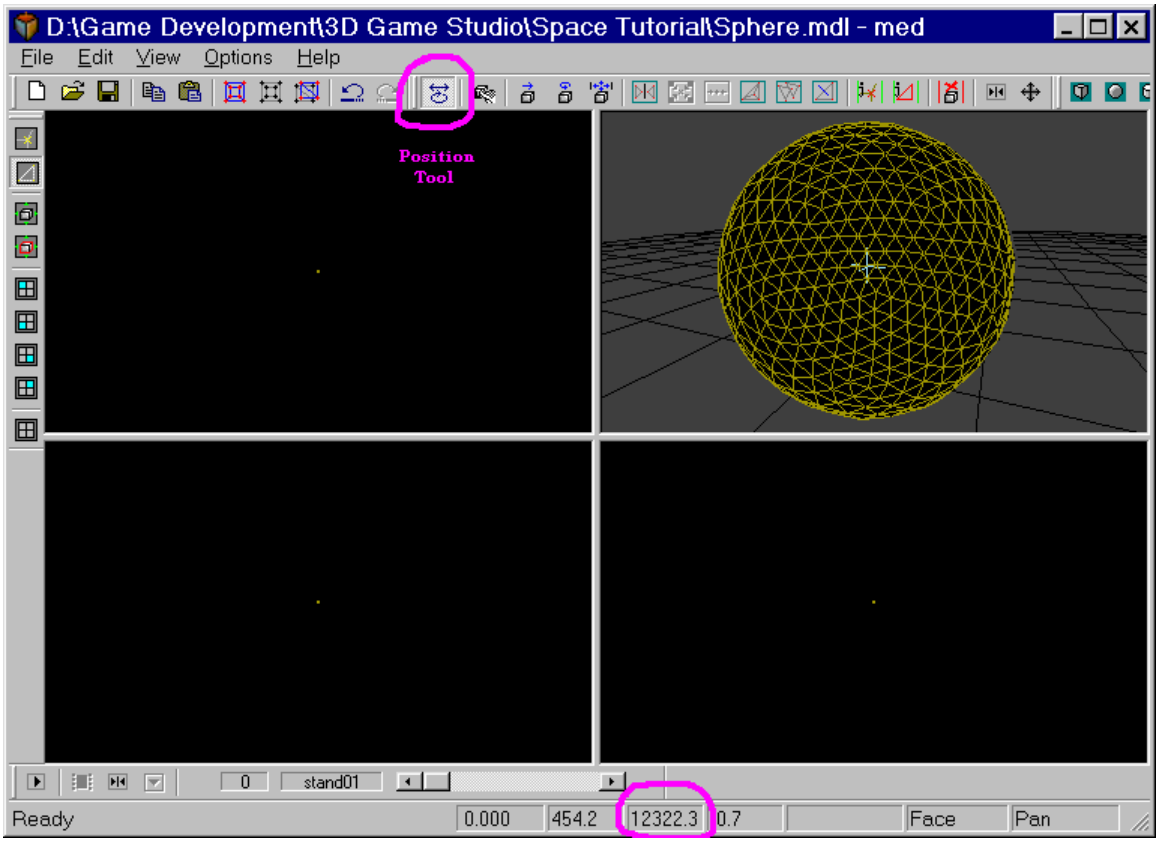

Abbildung 5: Kamera in Position zum skalieren der Kugel

Als nächstes wählen Sie das Skalierungswerkzeug (vierter Knopf rechts neben dem Positionsknopf, s. Abb. 6). Die Kugel sollte immer noch als Ganzes markiert sein. Jetzt klicken Sie einfach mit der linken Maustaste in einem der 2D-Fenster und ziehen den Mauspfeil nach unten bis die Kugel die gesamte Fläche ausfüllt.

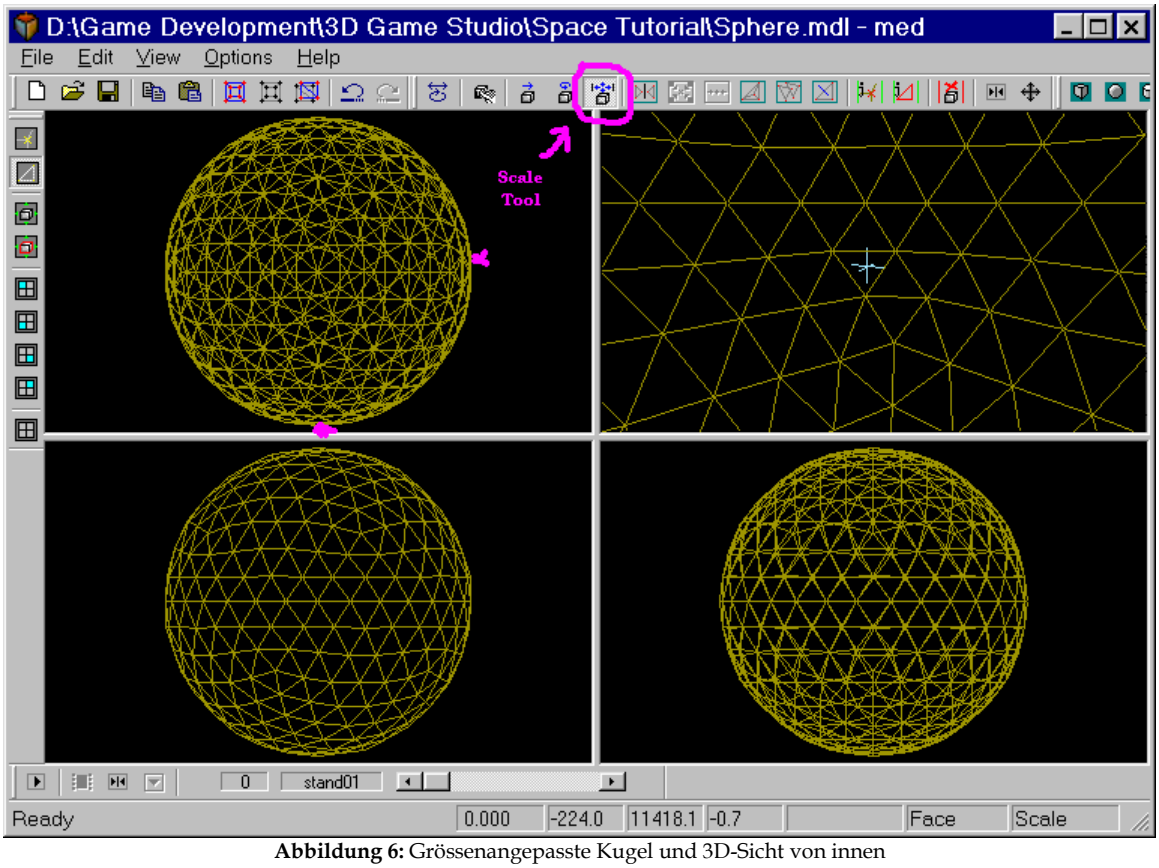

Abbildung 6: Grössenangepasste Kugel und 3D-Sicht von innen

Beachten Sie, dass sich die 3D-Ansicht nun im Innern der Kugel befindet! Wir haben also schon etwas erreicht!

Achten Sie auch auf die beiden Farbklekse, mit denen ich im Draufsicht-Fenster die rechte und untere Kante der Kugel markiert habe. Wenn Sie die Maus in etwa an diese Punkte bringen, können Sie anhand der Koordinatenanzeige am unteren Bildschirmrand recht gut einschätzen, wie gross Ihre Kugel geworden ist. Die genauen Abmessungen sind nicht so entscheident, sie muss einfach nur relativ gross sein. Nachdem Sie das Ganze im Level ausprobiert haben, können Sie jederzeit zu diesem Punkt zurückkehren und die Grösse erneut anpassen,.

Step 4: Oberfläche aufspannen!

Der letzte Schritt zur vollendeten Kreation einer Sternenkugel ist der, ihr eine Oberfläche zu geben. Für unser Beispiel benutze ich, um Zeit zu sparen, ein MDL-Skinmapping. Je nachdem, wie Sie Ihr Himmelszelt angelegt haben, könnten an den Nahtstellen zwischen vorn und hinten deutliche Verzerrungen der Sterne auftreten. Die Verwendung von MD2-Skinnmapping erlöst uns von dem Problem und bringt das in Ordnung.

Die Himmelskugel, **`stars.mdl`**, die ich diesem Workshop beigefügt habe benutzt MD2-Skinmapping. Wenn Sie mehr darüber erfahren möchten, schlage ich vor, Sie schauen sich auf dem Conitec-User Forum nach **MD2 Skin** um oder Sie gehen im Internet z.B. über die Conitec-Link Seite auf einige der exzellenten Skinmapping-Tutorials.

So generieren wir unsere MDL-Skinmap:

1) Öffnen Sie MED. Klicken Sie aufs `view`-Menü und wählen Sie `skins` um den Skin-Editor zu öffnen.

2) Im Skin-Editor gehen Sie über `edit` auf `resize skin`. Im Dialogfenster setzen Sie die Grösse der Bilddatei fest, die Sie für Ihre Skin benutzen wollen. Ich habe in unserem Beispiel 512x512 verwendet. (Einige ältere Videokarten, wie Voodo2 verlangen möglicherweise eine kleinere Bitmap von max. 256x256. Probieren Sie das aus, falls Sie eine Fehlermeldung bezüglich Videospeicher kriegen).
3) Klicken Sie auf `edit`, dann `create mdl mapping`. Im Dialogfenster wählen Sie `front` und `ok`. Die Polygon-Map sollte nun aufgeklappt als Vorder- und Hinterteil des Kugelmodels wie in Abbildung 7 auf dem Bildschirm zu sehen sein.

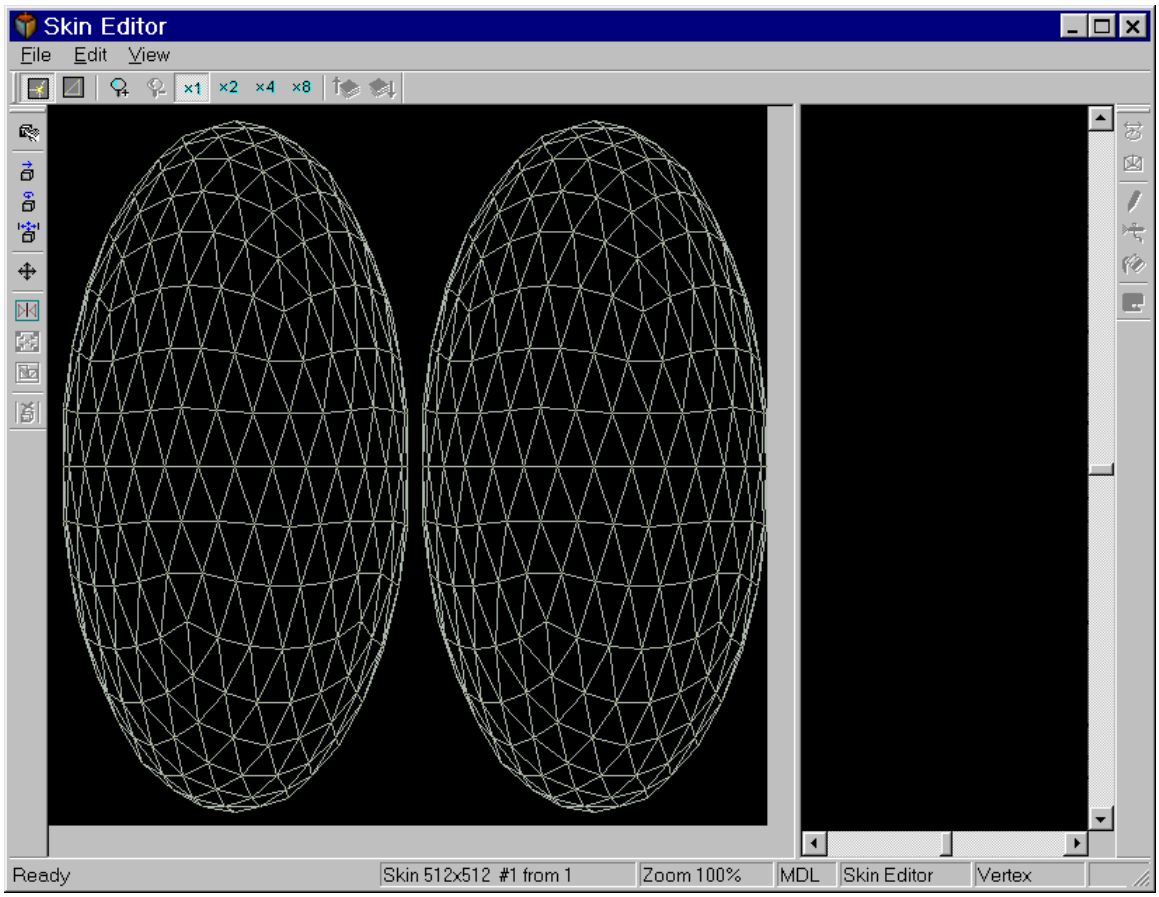

Abbildung 7: Exportieren der MDL-Oberflächenmap

4) Klicken Sie auf `file`,dann `export` und anschliessend, je nachdem, was Sie bevorzugen, entweder `export to bmp` oder `pcx`.
5) Speichern Sie Ihre Oberflächenhaut (Skin Map).
6) Öffnen Sie die Skin Map in einem Malprogramm Ihrer Wahl. Halten Sie die Bitmap hauptsächlich in Schwarz und streuen Sie ein paar Sterne und andere Himmelkörper ein. (Ich verwende hier die mitgelieferte **`starmap.pcx`** -Datei).
7) Speichern Sie die neue Skin Map.
8) Gehen Sie zurück zum Skin Editor und klicken dort auf `file import` → `skin image`. Nehmen Sie Ihr soeben gemaltes Bild. Nun sollten Sie etwas haben, das in etwa so aussieht, wie in Abbildung 8.

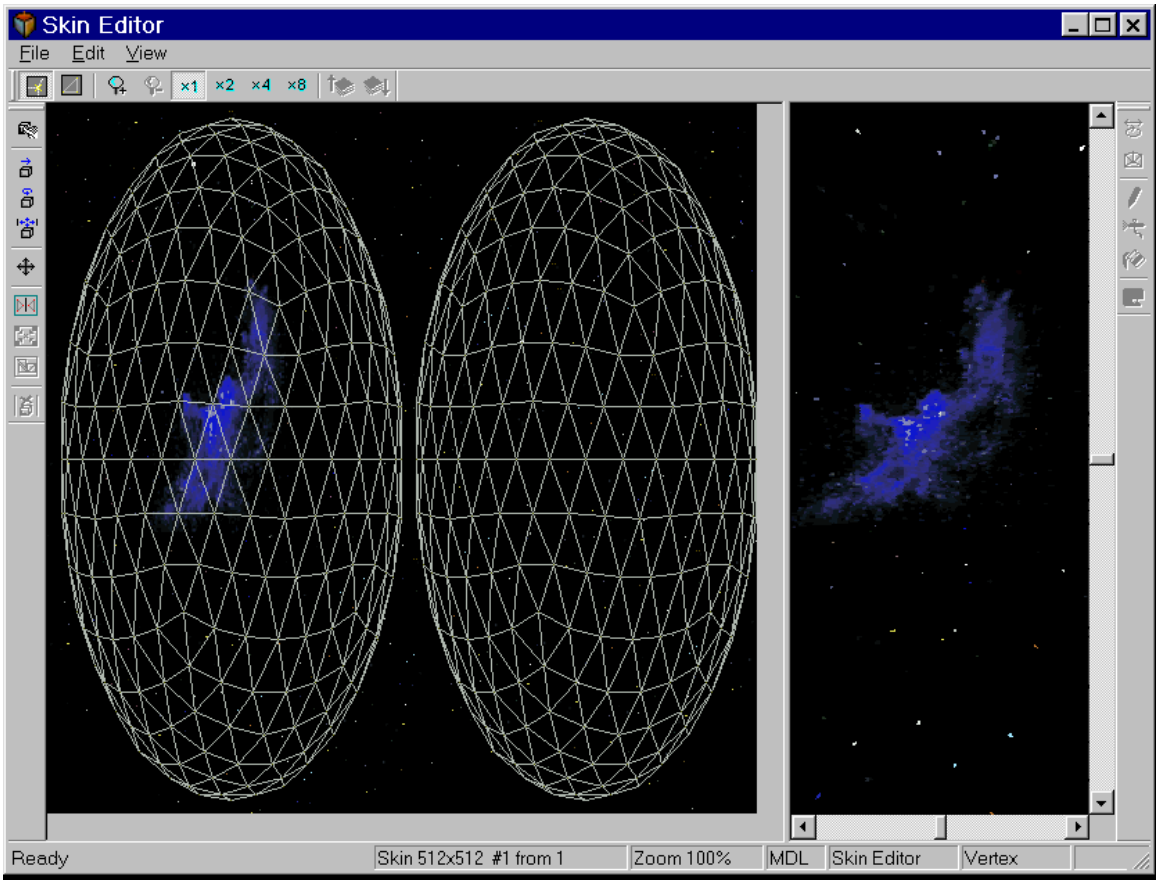

Abbildung 8: Eine fertige All-Oberfläche

Bedenken Sie, dass dies eine grob hingehuschte 'quick and dirty' Sternenkugel ist. Die Dreiecke des Drahtgitters sind offensichtlich von unterschiedlicher Grösse und die Sterne des flachen Feldes zwischen den Hälften sind nicht mit auf die Kugel aufgespannt. Für eine richtig gute Kugel bräuchten wir mehr Zeit und würden eine zylindrische Projektion erstellen, wie bei der Darstellung der Weltkugel im Atlas. Doch das All ist dunkel und verdeckt die Löcher und Flicken unseres Mappings taktvoll. Schliessen Sie den Skin-Editor und bewundern Sie Ihr Werk mit Hilfe des Positions-Knopfes in der 3D-Ansicht des WED!

Nun speichern Sie Ihr fertiggestelltes Sternenmodel noch per `file` → `save as` in den "**Space**"-Ordner. Nennen Sie es **`mystars.mdl`**

Peng! Sie haben soeben ***Ihr*** Universum erschaffen!

Zusammenfassung:

1) Erstelle eine Kugel
2) Klappe die Normalen um
3) Skaliere sie
4) Gib ihr eine Oberfläche!

Erstellen des All-Levels

Tatsächlich besteht das All als Level aus nichts weiter, als einer unsichtbaren zurückfedernden Umhüllung, der sogenannten "Bounding Box", mit unserem Raumschiff drin. Der Blick auf die Sterne hängt dabei von der Position unseres Schiffs ab. Aus diesem Grund wird unsere Sternenkugel vom Skript erst generiert, wenn das Synonym des Raumschiffs gesetzt ist. So wird der Himmel, wenn das Level einmal gestartet ist, automatisch überwacht und bewegt sich mit dem Schiff mit. Damit vermitteln wir den Eindruck von den unendlichen Weiten des Weltraums.

Um das All-Level zu erstellen, öffnen Sie WED und wählen `file` →`new`.

Fügen Sie Ihrem Level eine WAD-Datei hinzu: `texture` → `texture manager`

Klicken Sie auf den `add wad`-Knopf, suchen Sie sich eine WAD Ihrer Wahl heraus und öffnen diesen (`open`, s. Abb. 9). Ich verwende die **`standard.wad`**-Datei.

Wenn Sie fertig sind, schliessen Sie das Fenster des Textur-Managers.

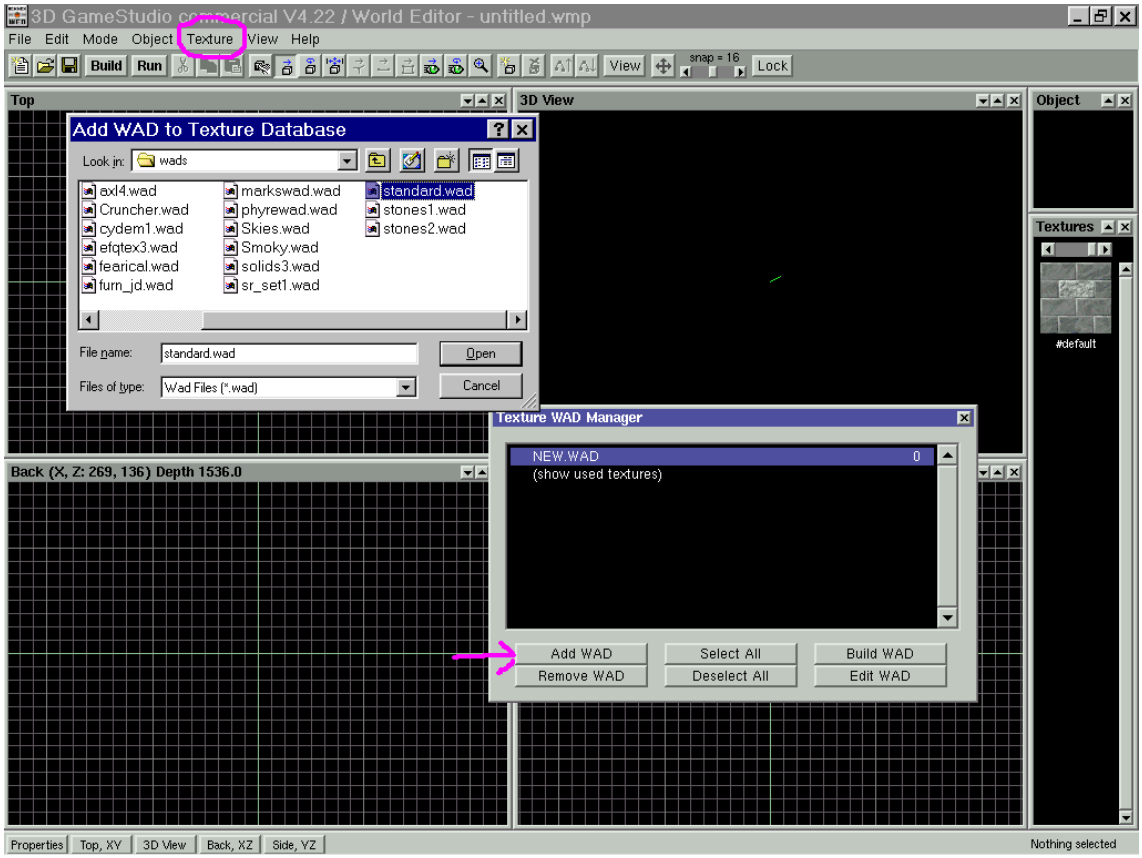

Abbildung 9: Dem Level eine WAD zuweisen

Als nächstes erstellen wir unsere Bounding Box. Tun Sie dies, indem Sie `object` → `add primitive` →`cube large` auswählen.

Skalieren Sie den Würfel auf etwa 30.000 Quants pro Seite.

Das machen Sie am besten, indem Sie die Blicktiefe in allen Ansichtsfenstern mit der [+]-Taste soweit wie nötig erhöhen. In unserem Fall erweitern Sie also die Tiefe in jedem Fenster auf 30080 (s. Abb. 10).

Nun aktivieren Sie `zoom eye` oben in der Hauptwerkzeugsleiste (s. Abb.10). Klicken Sie links in einem der 2D-Fenster und ziehen Sie die Maus nach unten. Dadurch kriegen Sie die Kamera sehr

weit weg und Sie erhalten einen guten Überblick um das Level korrekt zu skalieren (ich habe bis zum Anschlag weggezoomt). In dem Moment erscheinen Ihre 2D-Würfel als kleine, unscheinbare Pünktchen.

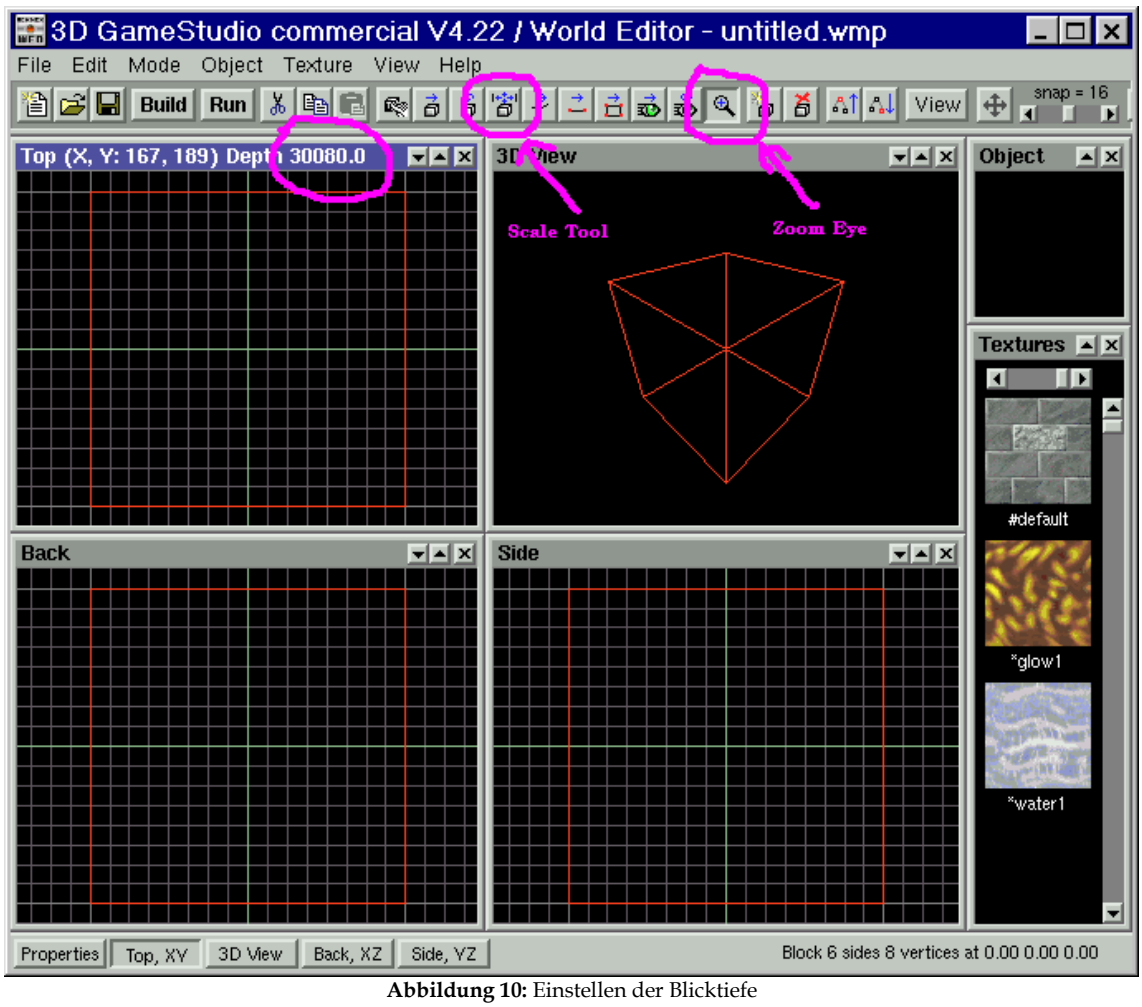

Abbildung 10: Einstellen der Blicktiefe

Gehen Sie dann auf das Skalierungswerkzeug (`scale`-tool Abb.10) und skalieren Sie den Würfel in jedem Fenster auf 30.000 Quants. (So gross es geht und er in den jeweils anderen Fenstern immer noch sichtbar ist). Das sollte dann wie in Abbildung 11 aussehen.

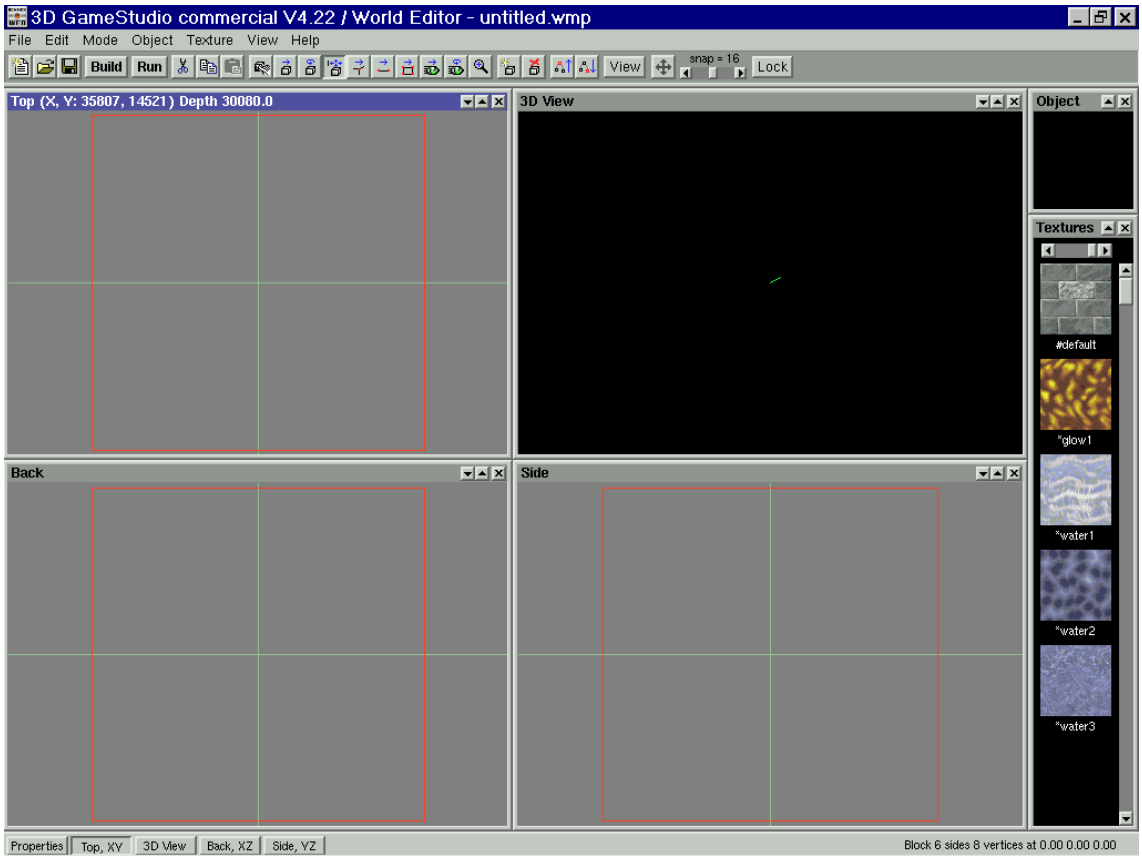

Abbildung 11: Skalieren eines Würfels auf 30.000 Quants

Jetzt höhlen wir diesen Würfel mit dem [**Alt / H**]-Tastaturbefehl aus. (Oder sie gehen mit der Maus übers Menü: `edit` → `hollow block`).

Beachten Sie, dass wir unserer Bounding Box nicht die Maximalgrösse von [ƒ]50.000 Quants gegeben haben. Und zwar deshalb, weil die Position unseres Raumschiffs immer im Zentrum unserer Sternenkugel bleibt und diese erstreckt sich über etwa 12.000 Quants.

Wenn unser Schiff bei 30.000 Quants gestoppt wird und unsere Sterne sich noch weitere 12.000 Quants ausbreiten, haben wir 42.000 Quants belegt... im sicheren Bereich unter der maximalen Levelgrösse. Sie wollen ja sicher gehen, dass nicht irgendwelche Teile Ihrer Sternenkugel die magische 50.000 Quants-Grenze überschreiten und Probleme bereiten.

Geben Sie der Bounding Box die Standardtextur, indem Sie die Textur mit der rechten Maustaste anklicken und anschliessend mit Linksklick `settings` wählen. Vergewissern Sie sich, dass die Standardtextur das **`flat`**-Flag gesetzt hat, ehe Sie diese zuweisen (s. Abb. 12).

ƒ In der neuesten Version A5 wurde diese Grenze auf 100.000 Quants erhöht

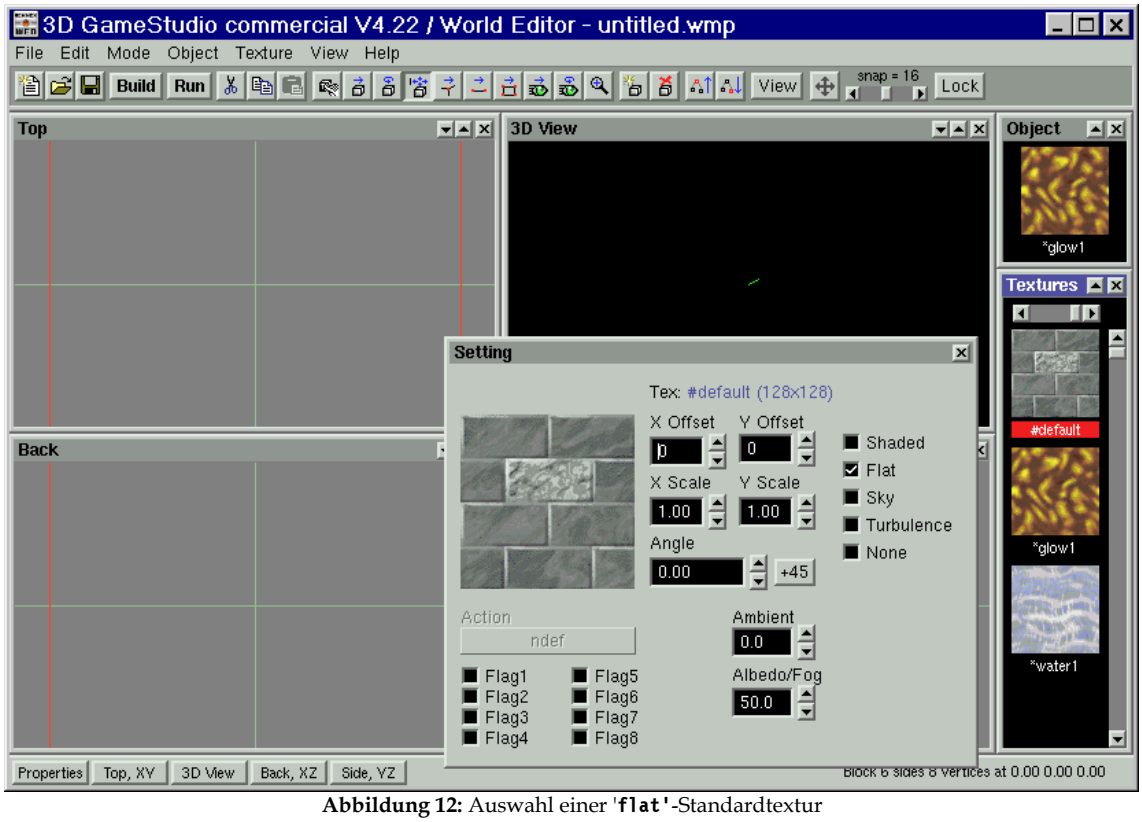

Abbildung 12: Auswahl einer '**flat**'-Standardtextur

Jetzt, da Ihre Aussenbox texturiert ist, öffnen Sie ihr Eigenschaftsfenster indem Sie den Properties-Knopf in der Bildschirmecke unten links anklicken. Achten Sie darauf, dass **invisible** der einzige gesetzte Flag ist (s. Abb. 13).

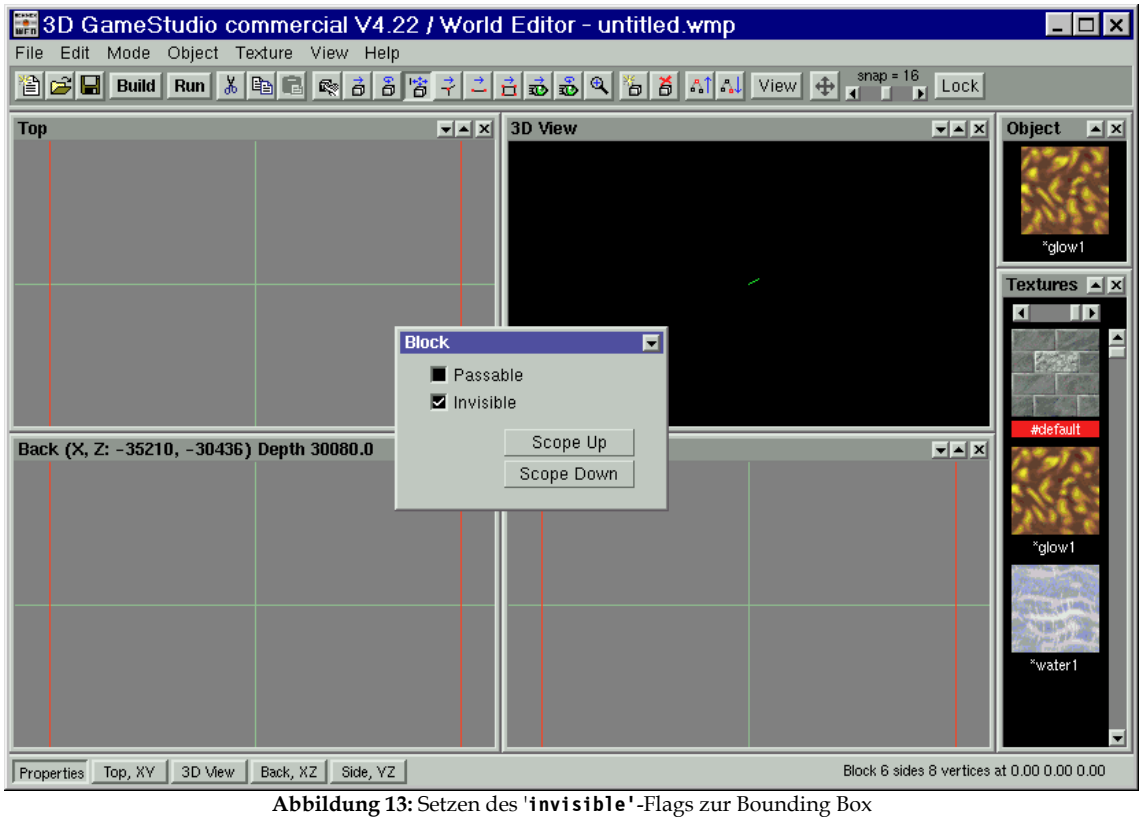

Abbildung 13: Setzen des '**invisible**'-Flags zur Bounding Box

Gratulation! Sie haben soeben die Bounding Box vollendet – die Grenze des bekannten Universums!

Zoomen Sie nun wieder näher heran und fügen Sie mit `object` → `add primitive` ein paar einfache Bausteine in Ihr Level ein und texturieren Sie diese nach eigenem Geschmack.

Grösse, Position und Texturen sind nicht von grundsätzlicher Bedeutung, es macht sich aber besser, wenn sie einige Objekte verteilen. Vor allem ist es wichtig, dass Sie ein paar in die Nähe der zukünftigen Schiffsposition setzen. Das gibt dann das Gefühl, dass das Schiff sich fortbewegt. Es ist nunmal ein leidiges Problem, dass das All ziemlich leer ist. Ohne stationäre Bezugspunkte ist es unmöglich festzustellen, ob man sich nun bewegt, oder nicht und man kann leicht verloren gehen.

Nun fügen Sie dem Level mit noch `file` → `map properties` das **`spaceship.wdl`**-Skript hinzu (s. Abb. 14).

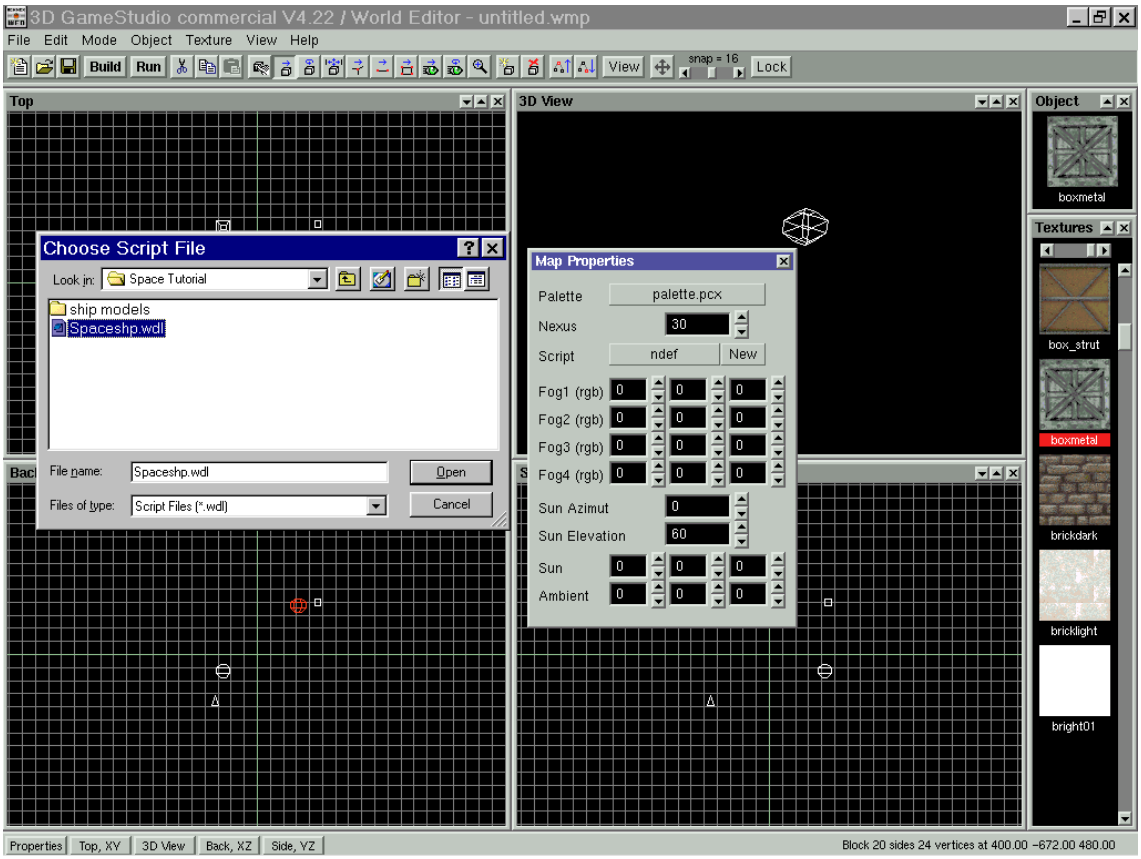

Abbildung 14: Hinzufügen der '**`spaceshp.wdl`**'

Wir sind schon auf der Zielgeraden!

Geben Sie jetzt nur noch die Kamera-Startposition an: `object` → `add start position.` Rücken Sie die Kamera ein bisschen zur Seite, aber mit Blickrichtung zum Zentrum (s.Abb.15).

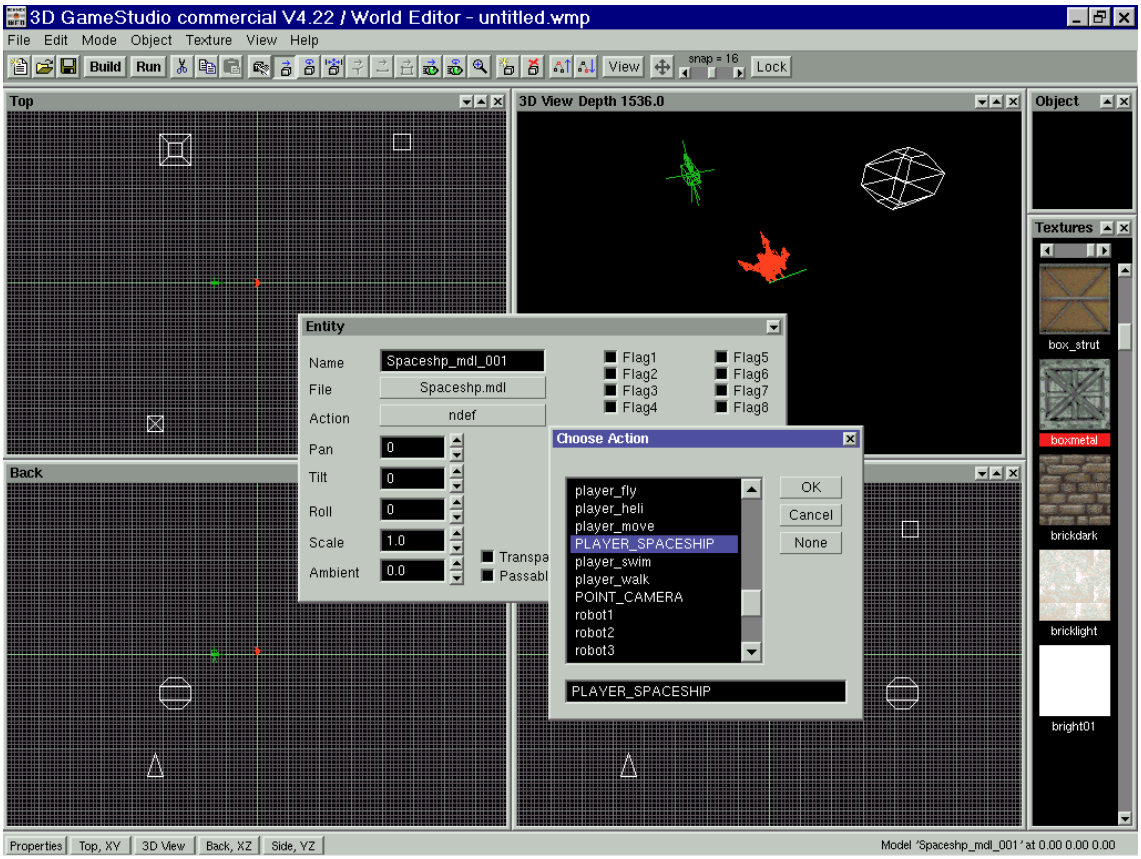

Abbildung 15: Einfügen von Kamera und Raumschiff

Das einzige, was wir nun noch zur Vervollständigung unseres Weltraum-Levels tun müssen, ist ein Raumschiff einzufügen!

Machen Sie das mit `object→load entity` und suchen Sie im Browser-Fenster nach dem **`spaceshp.mdl`**-Model , welches Sie dann ins Level einfügen. Plazieren Sie das Schiff im Zentrum und achten Sie darauf, dass Ihre Kamera auch wirklich darauf gerichtet ist.

Mit Rechtsklick aufs Raumschiff öffnen Sie sein Eigenschaftsfenster. Klicken Sie auf den `action`-Knopf und skrollen Sie durch die Liste bis Sie **`player_spaceship`** markieren können. Klicken Sie auf `ok`.

Glückwunsch! Soeben haben Sie die Erschaffung Ihres ersten Weltraums abgeschlossen!

Speichern Sie das Level in Ihrem "**Space**"-Ordner und nennen Sie es ruhig **`bigspace.wmp`**

`build`en Sie das Level mit markierter Level-Map und starten Sie es dann.

Wenn Sie alles richtig gemacht haben, sollten Sie nun Ihr Raumschiff durchs All gleiten sehen!

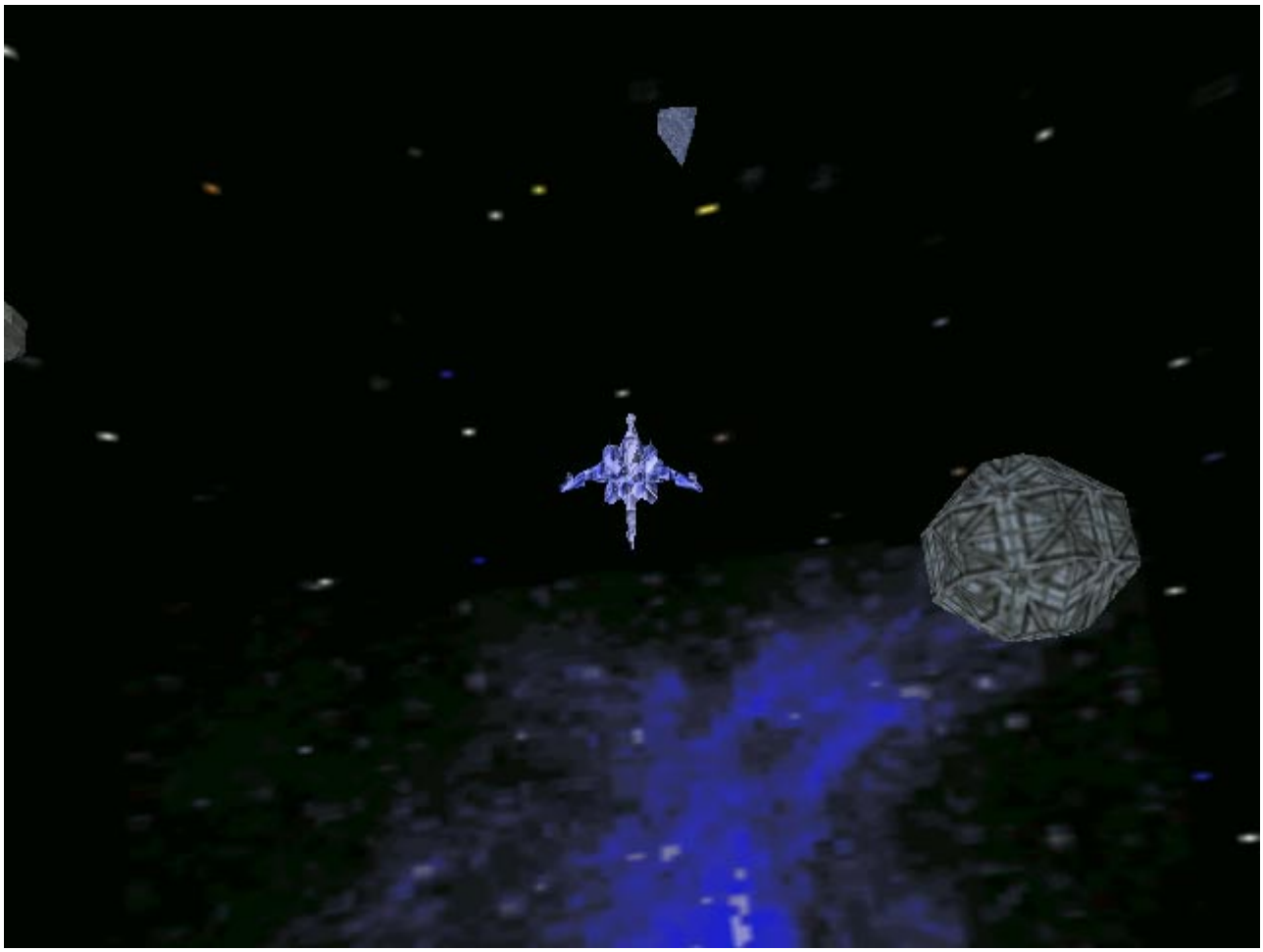

Abbildung 16: Unser fertiges Space- Level!

Spielen Sie ein bischen mit Ihrem Raumschiff herum. Die Bedienung ist einfach:

- Mit den [🡐] / [🡒]-Pfeiltasten drehen Sie das Schiff
- Mit den [🡑] / [🡓]-Pfeiltasten kontrollieren Sie die Neigung
- Mit der [**Leertaste**] zünden Sie die Antriebsdüsen

Individuelles Anpassen des Raumflugs

Wenn Sie sich einigermassen mit dem Raumschiff und seiner Bedienung vertraut gemacht haben, machen Sie ein **Backup Ihres `spaceshp.wdl`-Skripts** (unter anderem Namen abspeichern). Versuchen Sie sich nun an einigen Änderungen der Parameter und Variablen, die in der **`player_spaceship`**-Aktion Ihres **`spaceshp.wdl`-**Skriptes gesetzt sind.

Öffnen Sie das **`spaceshp.wdl`-**Skript in Ihrem bevorzugten Texteditor und modifizieren Sie die unten aufgelisteten Variablen so, dass Sie Ihren eigenen Bedürfnissen gerecht werden. Vergessen Sie nicht, das Level zu Speichern. Danach starten Sie Ihr Level und betrachten sich Ihr Werk in Aktion.

Sie können die folgenden Variablen im Skript ändern:

camera_type

Ist zunächst auf **1**, den "**Chase Cam**"-Modus, gesetzt. Dabei wirkt das Schiff stationär und die Welt drumrum in Bewegung. Wird diese Variable auf **2** gesetzt, sehen Sie den Blick aus dem Cockpit und auf **3** können Sie Ihr Schiff vom Blickwinkel einer ausserhalb fixierten Kamera aus betrachten.

my.auto_spin_stop
Ist zunächst auf **1** gesetzt, was Ihre Rotation automatisch verlangsamt, solange Sie die Steuerdüsen nicht aktiv einsetzen. Ist diese Variable auf **0**, wird Ihr Schiff solange rotieren, bis Sie manuell mit den Steuerdüsen aus der anderen Richtung gegensteuern.

my.auto_decel
Ist zunächst auf **1** gesetzt, was dem Schiff ein Anhalten erlaubt. Wird diese Varible auf **0** gesetzt, so driftet das Schiff entlang seines gegenwärtigen Vectors weiter.

my.limit_turn_speed
Ist zunächst auf **1** gesetzt, was die Rotationsgeschwindigkeit in Abhängigkeit von der **my.max_spin_speed**-Variablen begrenzt. Ist der Wert auf **0**, gibt es keine Geschwindigkeitsbegrenzung der Drehung.

my.limit_top_speed
Ist dieser Wert auf **1**, wird jedem Vektor der Steuerdüse eine Geschwindikeitsbegrenzung beigegeben. Diese wiederum wird vom Wert von **my.max_ship_speed** vorgegeben. Ist die Variable auf **0**, gibt es auch keine Geschwindigkeitsbegrenzung.

my.engine_thrust
Definiert die wirksame Kraft des "**Hauptantriebs".** Höhere Werte geben mehr Schub. Der Standardwert ist **.5**.

my.spin_rate
Definiert die wirksame Kraft der "**drehenden Düsen**". Höhere Werte ergeben eine schnellere Drehung (und erschweren die Kontrolle über das Schiff!). Der Standardwert ist .**15**.

my. decel_rate
Definiert die auf das Schiff im Raum wirkende **"Reibung**". Ich weiss, das klingt nicht gerade sehr realistisch, deshalb können Sie diese auch ausser Kraft setzen, indem Sie den **my.auto_decel**-Flag auf **0** stellen. Allerdings ist das Schiff mit einer gewissen Reibungskraft sehr viel leichter zu handhaben. Der Standardwert liegt bei **.04**.

my. max_spin_speed
Definiert die Höchstgeschwindigkeit bei der die "**drehenden Düsen**" noch zünden. Höhere Werte erlauben eine schnellere maximale Drehgeschwindigkeit des Schiffes. Wenn Sie den Flag **my.limit_turn_speed** auf **0** setzen, ist diese Option ausser Kraft. Der Standardwert ist **5**.

my. max_ship_speed
Definiert die wirksame Höchstgeschwindigkeit für jeden Vektor der Schiffsbewegung. Höhere Werte erlauben dem Schiff schnellere Höchstgeschwindigkeiten. Sie können diese Option abschalten, wenn Sie den **my.limit_top_speed**-Flag auf **0** setzen. Der Standardwert ist **20**.

Zum Schluss!

Soweit der Inhalt dieses Raumflugworkshops. Ich hoffe, es hat Ihnen gefallen! Dieser Kurs bietet natürlich nur eine erste Grundlage für Spiele, die im Weltraum angesiedelt sind. Er wurde auf Basis einer recht realistischen Physik konstruiert, und es kann eine Menge Spass machen, damit zu arbeiten. Es bleibt aber auch noch viel Raum für Verbesserungen. Wenn dieser Workshop auch ein hervorragendes Sprungbrett für Weltraum-Simulationsspiele ist, wollen Sie vielleicht lieber mit etwas weniger Realismus und dafür ein bisschen mehr Kontrolle ein Arcade-Weltraumspiel erstellen? Ich bin sehr gespannt darauf, was andere präsentieren werden!

Hier ein paar Ideen, womit Sie Ihre extra terrestrischen Aktivitäten noch ein wenig aufpeppen könnten: bauen Sie ein Panel fürs Cockpit ein (falls Sie in dieser Ansicht operieren!), nehmen Sie Waffen ins Level, fügen Sie Befehle zum Kontrollieren der Rollbewegungen Ihres Raumschiffs ein, machen Sie schönere Sternen-Maps, hübschere Features (inklusive Planeten!) in WED. Dann wären da noch Lande-, Andock-, Abhebesequenzen...

Und sollten Sie ein ungewöhnliches und interessantes Flugprofil für ein Raumschiff entworfen haben, veröffentlichen Sie es im Forum, so dass andere auch etwas davon haben!

Noch ein Letztes... Schauen Sie nach dem "**Vertex Space Tool**", das es demnächst auf der Conitec-Download-Seite geben wird. Bei diesem Werkzeug handelt es sich um ein Interface, welches auf einem Panel basiert und damit haben Sie dann die Möglichkeit alle Flags und Einstellungen wörtlich "im Fluge" zu verändern!

Der Himmel kennt keine Grenzen mehr!

- Nick "WildCat" Chionilos

Für die Wagemutigen, die unserem Raumschiff unter die Motorhaube schauen und wissen wollen, wie es denn nun funktioniert, gibt es im **Anhang** das detailliert kommentierte Skript. Es setzt einigermassen solide Grundkenntnisse der WDL voraus und, an manchen Stellen, ein wenig Mathematik - oder Sie glauben mir eben einfach!

APPENDIX : Einzelheiten zum SPACESHP.WDL-Skript

```
#                    Willkommen zum Weltraum-Vorlage!
#
#       Die Steuerung ist denkbar einfach:
#       Mit den Pfeiltasten [links] / [rechts] drehen Sie das Schiff
#       Mit den Pfeiltasten [rauf] / [runter] steuern Sie die Neigung des Schiffs
#       Mit der [Leertaste] kontrollieren Sie die Antriebsdüsen
#
#    ACHTUNG: Optionen, die vom Anwender zu ändern sind, sind fett und blau - wie das hier -
#
////////////////////////////////////////////////////////////////////////////
//      A4 main (Haupt)- wdl
////////////////////////////////////////////////////////////////////////////
// Das PATH (Pfad)-Schlüsselwort gibt - relativ zum Levelverzeichnis - das Verzeichnis an,
// in dem die entsprechenden Dateien gespeichert sind
path    "models";               // Pfad zum Modelunterverzeichnis - falls es eines gibt
path    "sounds";               // Pfad zum Sounduntervezeichnis - falls es eines gibt
path    "bmaps";                // Pfad zum Grafikunterverzeichnis - falls es eines gibt
path    "images";               // Pfad zum Bilderunterverzeichnis - falls es eines gibt
path    "..\\template";         // Pfad zum WDL-Template-Unterverzeichnis


////////////////////////////////////////////////////////////////////////////
// Das INCLUDE-Schlüsselwort wird zum Einfügen weiterer WDL-Dateien verwendet,
// wie z.B. solche mit vorgefertigten Aktionen, wie im TEMPLATE-Unterverzeichnis.
////////////////////////////////////////////////////////////////////////////

include <movement.wdl>;
include <messages.wdl>;
include <particle.wdl>;
include <doors.wdl>;
include <actors.wdl>;
include <weapons.wdl>;
include <war.wdl>;
include <menu.wdl>;

// Setze die initiale Bildschirmauflösung fürs System
ifdef lores;
var video_mode = 4;             // 320x240
ifelse;
var video_mode = 6;             // 640x480
endif;

var video_depth = 16;           // D3D, 16 bit-Auflösung
var fps_max = 40;               // 40 fps max


// Die MAIN-Funkton wird bei Spielstart aufgerufen und bringt alles zum laufen

function  main()
{
        load_level <bigspace.wmb>;
        load_status();          // globale variablen laden
        wait 16;
}
```

```
////////////////////////////////////////////////////////////////////////////
//      BEGINN DES SPACE SKRIPTS:
////////////////////////////////////////////////////////////////////////////

// Sounds:

        sound main_engines,<engine.wav>;
        sound thrusters,<thruster.wav>;
        var thrust_handle = 0;   // die Sound-handles kontrollieren
        var engine_handle = 0;   // wann ein Sound abgespielt wird

// Synonyms:

        synonym player_ship {type entity;}
        synonym star_sphere {type entity;}

//      Der ACCEL_VECTOR drückt aus, in welche Richtung das Schiff vom Schub beschleunigt wird.
//      Weil das Schiff so gebaut wurde, dass sein Bug entlang der X-Achse zeigt, wirkt
//      der Antriebsschub immer auf den "X"-Komponenten von ACCEL_VECTOR. Zur Vereinfachung
//      habe ich diesen darum in  ENGINE_THRUST ('Maschinenschub') umbenannt.
//

        define accel_vector,skill1;
        define engine_thrust,skill1;


//      Während ACCEL_VECTOR den Schub relativ zum Raumwinkel des Schiffes bestimmt,
//      ist der für die Geschwindigkeit in Relation zum Level zuständig.

        define drift_vector,skill4;
        define drift_vector_x,skill4;
        define drift_vector_y,skill5;
        define drift_vector_z,skill6;

        define ship_angles,skill7;       // Pan-, Tilt- und Roll-Geschwindigkeiten des Schiffs
        define pan_speed,skill7;         // So schnell drehen Sie sich um die Z-Achse (re / li)
        define tilt_speed,skill8;        // So schnell drehen Sie sich um die Y-Achse (Neigung auf / ab)
        define roll_speed,skill9;        // So schnell drehen Sie sich um die X-Achse (Längsachse d. Schiffs)
        define spin_rate,skill10;        // Die Beschleunigung Ihrer Drehbewegung

// *** ACHTUNG: Auch wenn der Koeffizient Reibung im All praktisch **NULL** ist, macht es
//      das Schiff doch sehr viel leichter steuerbar und das Spieleverhalten lässt sich besser
//      einstellen, wenn man DECEL_RATE einen kleinen Wert gibt, oder Geschwindigkeit und
//      Dreheung des Schiffes beschneidet. Stellen Sie sich einen Autopiloten vor,
//      der die geeigneten Schritte unternimmt, wenn das Schiff nicht aktiv gesteuert wird.
//      Diese Definitionen werden benutzt, wenn ihre enstprechenden Flags auf on gesetzt sind:

// Wie Schnell soll die Reibung "Friction" Sie abbremsen?
        define decel_rate,skill11;

// Begrenzung der Drehgeschwindigkeit, wenn der Flag LIMIT_TURN_SPEED gesetzt ist
        define max_spin_speed,skill12;

// Festlegen der Höchstgeschwindikeit, wenn der Flag LIMIT_TOP_SPEED gesetzt ist
        define max_ship_speed,skill13;


//*** Definitionen der Flags, die die Flugeigenschaften des Raumschiffs kontrollieren
        define auto_spin_stop,flag1;     //    Für absoluten Realismus gehören alle
        define auto_decel,flag2;         //    diese Flags abgeschaltet. Zur besseren Handhabung
        define limit_turn_speed,flag3;   //    des Schiffes sind sie aber trotzdem
```

```
        define limit_top_speed,flag4;    //    nützlich.

//define star_sky,<stars.mdl>;           // Standard Kugel-Model aus dem Tutorial(auskommentiert)
define star_sky,<mystars.mdl>;                  // Neues, vom Anwender selbst erstelltes Kugel-Model


// ************ Die aktiven Variablen beginnen hier ******************
var ship_angs[3];               // Wird für PAN- und TILT- Winkelberechnungen gebraucht
var engine_work_speed[3];       // Wird für Geschwindigkeits- und Abdriftsberechnungen gebraucht


// Kamera Variable
 var camera_type;
  var camera_spot[3];
  var workang[3];

// Die Himmelskugel bewegt sich einfach nur mit dem Raumschiff des Spielers mit.
//      Indem wir die X,Y und Z-Koordinaten direkt über seine POS-Variable verändern,
//      unterbinden wir jegliche Kollisionserkennung und das ist es, was wir wollen.
//       Die Sternenkugel soll nämlich durch die Bouncing Box hindurch und über das
//      Level hinausgehen können, das Raumschiff hingegen nicht!

action space_sphere
{
        star_sphere = me;

        while (1)       // Halte den Spieler konstant im Zentrum des Universums
        {
                vec_set(my.pos,player_ship.pos);
                wait 1;
        }
}

action player_spaceship
{
// Setze Kameramodus
        camera_type = 1;      //**** 1 = Chase Cam, 2 = Cockpit, 3 = Stationär ****

// Setze Synonyme für spätere Anwendung und setze sämtliche Schiffsgeschwindigkeiten ausser Kraft
        player_ship = me;
        my.narrow = on;                         // Verwende kleinere Kollisionshülle
        vec_set(my.accel_vector,nullvector);    // Keine Startgeschwindigkeit
        vec_set(my.drift_vector,nullvector);    // Keine Startdrift
        create star_sky,my.pos,space_sphere;    // Initialisiere Hintergrundsterne


//*********************************************************
// Alle speziellen Flug-Flags sind zum Starten aktiviert. Um sie abzuschalten, setzen Sie sie auf 0.

        my.auto_spin_stop = 1;      // Beendet die Eigendrehung des Schiffes
        my.auto_decel = 1;          // Bremst das Schiff mit Hilfe von  MY.DECEL_RATE ab
        my.limit_turn_speed = 1;    // Limitiert die Drehgeschwindigkeit in Abhängigkeit von
                                    // MY.MAX_SPIN_SPEED
        my.limit_top_speed = 1;     // Höchstgeschwindigkeit des Schiffs, abhängig von MY.MAX_SHIP_SPEED

// Flugeigenschaften des Schiffes beim Starten

        my.engine_thrust = .5;              // Stärke der Haupttriebwerke
        my.spin_rate = .15;                 // Stärke der Steuertriebwerke
        my.decel_rate = .04;                // Grad der Geschwindigkeitsverringerung der Drift.
        my.max_spin_speed = 5;              // Maximale Geschwindigkeit für Richtungswechsel des Schiffs.
        my.max_ship_speed = 20;             // Absolute Höchstgeschwindigkeit des Schiffs
```

```
//*********************************************************

        while (1) {

# *****Logik des Dreh-Schubs:
#
#       Wenn Sie ein Triebwerk zünden und LIMIT_TURN_SPEEDS verwenden, vergewissern Sie sich,
#       dass Sie dabei immer noch unter der Höchstgeschwindigkeit liegen. Machen Sie das
#       für die die Pfeiltasten Links (CUL), Rechts (CUR), Auf (CUU), und Ab (CUD).

                if (key_cul == 1)
                    {if ((my.limit_turn_speed == 0) ||
                         (my.limit_turn_speed == 1) &&
                         (my.pan_speed < my.max_spin_speed))
                        {my.pan_speed += my.spin_rate * time;
                          if (thrust_handle == 0)               // kein Sound wird abgespielt
                               {play_loop thrusters,15;         // Spiele den Triebwerks-Sound
                            thrust_handle = result;}}}

                if (key_cur == 1)
                    {if ((my.limit_turn_speed == 0) ||
                           (my.limit_turn_speed == 1) &&
                           (my.pan_speed > -my.max_spin_speed))
                        {my.pan_speed -= my.spin_rate * time;
                          if (thrust_handle == 0)               // kein Sound wird abgespielt
                               {play_loop thrusters,15;         // Spiele den Triebwerks-Sound
                            thrust_handle = result;}}}

                if (key_cuu == 1)
                    {if ((my.limit_turn_speed == 0) ||
                           (my.limit_turn_speed == 1) &&
                           (my.tilt_speed < my.max_spin_speed))
                        {my.tilt_speed += my.spin_rate * time;
                          if (thrust_handle == 0)               // kein Sound wird abgespielt
                               {play_loop thrusters,15;         // Spiele den Triebwerks-Sound
                            thrust_handle = result;}}}

                if (key_cud == 1)
                    {if ((my.limit_turn_speed == 0) ||
                           (my.limit_turn_speed == 1) &&
                           (my.tilt_speed > -my.max_spin_speed))
                        {my.tilt_speed -= my.spin_rate * time;
                          if (thrust_handle == 0)               // Wenn kein Sound abgespielt wird,
                               {play_loop thrusters,15;         // dann spiele den Triebwerks-Sound
                            thrust_handle = result;}}}

                if ((thrust_handle != 0) &&      // Wenn ein Sound gespielt wird...
                        (key_cuu == 0) &&
                        (key_cud == 0) &&        //      ...und...
                        (key_cul == 0) &&
                        (key_cur == 0))          //  keine Pfeiltasten gedrückt wurden...
                                {stop_sound thrust_handle;       // beende den Sound
                                 thrust_handle = 0;}

//******* Falls Auto_Spin_Stop benutzt wird, verlangsame die Drehung des Schiffes, sofern es nicht aktiv
// gesteuert wird.

                if (my.auto_spin_stop == 1)
                        {stop_rotation();}

//********Zünden der Haupttriebwerke:
```

```
                if (key_space == 1)             // falls die Leertaste gedrückt ist
                        {if (engine_handle == 0)        // falls kein Sound gespielt wird, dann spiele ihn ab
                         {play_loop main_engines,25;
                             engine_handle = result;}

//         1) Setze ENGINE_WORK_SPEED auf den gegenwärtigen Schub
//         2) Vec_Rotate wandelt den Schub in Relation zur Richtung des Schiffes um
//      (was nur die X-Achse betrifft) und überträgt ihn auf die X-,Y- und Z-Komponenten
//      relativ zur Ausrichtung der umgebenden Welt.
//
// Achtung: Die Schritte 1 und 2 liessen sich ebensogut unter Verwendung von Trigonometrie machen,
// indem man die Ausrichtung der Umgebungswelt mit den folgenden Zeilen umgesetzt hätte:
//
//         X:   ENGINE_WORK_SPEED[0] += (MY.ENGINE_THRUST * COS(MY.TILT)
//                           * COS(MY.PAN));
//         Y:   ENGINE_WORK_SPEED[1] += (MY.ENGINE_THRUST * COS(MY.TILT)
//                           * SIN(MY.PAN));
//         Z:   ENGINE_WORK_SPEED[2] += (MY.ENGINE_THRUST * SIN(MY.TILT));
//
//      Die Verwendung von VEC_ROTATE erscheint einfach ein wenig gefälliger

                        vec_set(engine_work_speed,my.engine_thrust);
                        vec_rotate(engine_work_speed,my.pan);

//      Wenn es keine Geschwindigkeitsbegrenzung gibt, aktiviere den Schub, gibt es aber ein Limit, dann setze
//      nur die Komponenten des Schubs, die nicht beschnitten sind.

                        if (my.limit_top_speed == 0)
                                {vec_add(my.drift_vector,engine_work_speed);}
                        else
                                {if (abs(engine_work_speed.x + my.drift_vector_x) < my.max_ship_speed)
                                     {my.drift_vector_x += engine_work_speed.x;}

                                 if (abs(engine_work_speed.y + my.drift_vector_y) < my.max_ship_speed)
                                     {my.drift_vector_y += engine_work_speed.y;}

                                 if (abs(engine_work_speed.z + my.drift_vector_z) < my.max_ship_speed)
                                     {my.drift_vector_z += engine_work_speed.z;}}}
                else
                        {stop_sound engine_handle ;       // beende den Sound
                         engine_handle = 0;}

//**** Setze alles in Bewegung
        my.roll_speed = 0;                      // Wir verwenden ROLL nicht zur aktiven Steuerung
ROTATE ME,MY.SHIP_ANGLESS,NULLVECTOR;   // Rotieren des Schiffes abhängig von unseren
                                        // Pan-/Tilt-Werten.

// Wenn die Triebwerke nicht aktiv gezündet werden und Auto_decel aktiviert ist, verlangsame das Schiff
                if ((key_space != 1) && (my.auto_decel == 1))
                        {call decel_ship;}

// Berechne einen Abstand aus der Geschwindigkeit

vec_set(temp,my.drift_vector);
vec_scale(temp,time);
                move(me,nullvector,temp); // Dann bewege das Schiff gemäss der Drift

                point_camera();  // Justiere die Kamera
                wait 1;}
}

function stop_rotation()
{
```

```
//   **** Wenn Du immer noch rotierst, bringe zum Abbremsen eine Rotation in die entgegengesetzte
//       Richtung an.

        if (player_ship.pan_speed == 0)
            {goto checktilt;}
        else
            {if ((key_cul == 0) && (key_cur == 0))
                {if (player_ship.pan_speed > 0)
                    {player_ship.pan_speed -= player_ship.spin_rate * time;}
                 else
                    {player_ship.pan_speed += player_ship.spin_rate * time;}}
                 if (abs(player_ship.pan_speed) < .02)
                        {player_ship.pan_speed = 0;}}
checktilt:

        if (player_ship.tilt_speed == 0)
            {return;}
        else
            {if ((key_cuu == 0) && (key_cud == 0))
                {if (player_ship.tilt_speed > 0)
                    {player_ship.tilt_speed -= player_ship.spin_rate * time;}
                 else
                    {player_ship.tilt_speed += player_ship.spin_rate * time;}}
                 if (abs(player_ship.tilt_speed) < .02)
                        {player_ship.tilt_speed = 0;}}
}
}

# DECEL_SHIP funktioniert wie folgt:
# *** Wenn Schub da ist:
#    1) Setzen Sie ENGINE_WORK_SPEED damit die Verringerungsrate auf die "x"-Komponente des Schubs kommt
#               (Der Wert ist negativ, weil wir zum Abbremsen ja einen Gegenschub brauchen)
#    2) Finden Sie die zum bestehenden Drift-Vector passenden Pan- und Tilt-Winkel heraus,
#               verwenden Sie dafür Vec_to_angle.
#    3) Benutzen Sie Vec_Rotate, um den Schub von der Schiffsperspektive auf die Perspektive
#               der Umgebungswelt umzurechnen.
#    4) Verwenden Sie den Vec_Scale-Befehl zur Zeitkorrektur dieser Geschwindigkeiten bezüglich der Framerate.
#    5) Fügen Sie dem bereits vorhandenen Drift-Vektor zum Bremsen nun diesen neuen Verringerungs-Vektor hinzu.
#    6) Falls Sie sich nun doch noch sehr langsam ( < .02) weiterbewegen, setzen Sie Ihre Geschwindigkeit
#               (speed) auf Null.
# *** Sie sind zum Stillstand gekommen: tun Sie nichts und kehren Sie zurück (Return).

action decel_ship
{
        if (vec_length(player_ship.drift_vector) != 0)            // Wenn ich Schub habe
                {vec_set(engine_work_speed,nullvector);
                 engine_work_speed.x = -player_ship.decel_rate;
                 vec_to_angle(ship_angs.pan,player_ship.drift_vector);
                 vec_rotate(engine_work_speed,ship_angs.pan);
                 vec_add(my.drift_vector,engine_work_speed);}
        else
                {return;}               // Ist der DRIFT_VECTOR gleich Null, sind wir fertig

# *** Ist der DRIFT_VECTOR sehr nah bei Null, können Sie ihn bereits mit Null gleichsetzen. Dies verringert die
#       Anzahl der Durchgänge durch diesen Loop bis eine Entity dann keinen Impuls mehr hat.
#       Driftet eine Entity nicht, brauchen wir auch keine Mathematik für sie und können unserem Computer
#       daher die Arbeit ersparen. Das wiederum kann helfen die Framerate zu erhöhen, vor allem dann, wenn
#       viele Entities diese Routine zur selben Zeit verwenden.

        if (camera_type == 3)
                {goto stationary;}

// Chase Camera-Code:
//              (ACHTUNG: Der Chase Cam-Teil dieses Skripts ist eine Spende von JCL.
//               Gegenwärtig ist der Code noch einigermassen roh. Ich verstehe aber nicht genug davon, wie er
```

```
//              funktioniertIt um einen Erklärungsversuch unternehmen zu können. Bitte richten Sie Fragen
//              über diesen Abschnitt des Skripts an JCL selbst.)

        camera.genius = null;                   // die Kamera ignoriert niemanden
        vec_set(workang.pan,player_ship.pan);
        camera_spot.x = -200;                   // Relative Verschiebung der Kamera
        camera_spot.y = 0;                      //  zum Schiff, wenn im Chase-Modus
        camera_spot.z = 80;


        vec_set(camera.x,camera_spot.x);
        vec_rotate(camera.x,player_ship.pan);
        vec_add(camera.x,player_ship.x);

// Setze PAN- und TILT-Winkel der Kamera
        vec_set (tempa.x,player_ship.x);
        vec_sub (tempa.x,camera.x);
        vec_to_angle(camera.pan,tempa);



// Setze ROLL-Winkel der Kamera
        c1.x = tempa.y;
        c1.y = -tempa.x;
        c1.z = 0;

        c2.x = -camera_spot.y;
        c2.y = camera_spot.x;
        c2.z = 0;

        vec_rotate(c2,player_ship.pan);

// Errechne die Winkeldifferenz zwischen beiden Vektoren mit Hilfe des Skalarprodukts
        tempa = c1.x*c2.x +c1.y*c2.y;
        tempa /= vec_length(c1);
        tempa /= vec_length(c2);

        if (c2.z < 0)
                {camera.roll = acos(tempa);}
        else
                {camera.roll = -acos(tempa);}

        return;

//Cockpit-Kamera:
cockpit:
        camera.genius = player_ship;
        vec_set(camera.x,player_ship.x);        // Die Kamera ist da, wo das Schiff ist
        vec_set(camera.pan,player_ship.pan);    // Die Kamera schaut in dieselbe Richtung wie das Schiff.
        return;

//Stationäre Kamera:
stationary:
        camera.genius = null;
        vec_set(camera_work,player_ship.x);     // Richte die Kamera auf den Spieler
        vec_sub(camera_work,camera.x);
        vec_to_angle(camera.pan,camera_work);   // Nun schaut die Kamera auf den Spieler
        return;
}

//////////////  ENDE DES SPACE-SKRIPTS ///////////////////////
```