hairy hover abgabe 3:

kurzbeschreibung:

das programm wird durch doppelklick auf die datei spiel.exe im ordner /bin gestartet. zu beginn wird in der konsole ein spielmodus ausgewählt, wobei man das spiel in einem „fenster" spielen kann, oder im vollbildschirmmodus mit verschiedenen auflösungen. man muss also eine zahl zwischen 1 und 4 eingeben und enter drücken um das spiel im jeweiligen modus zu starten.

die steuerung:

- „enter" - start eines neuen spiels
- pfeiltasten für die richtung
- „a" und „s" um die kamera um das objekt zu drehen
- mit „d" wird das mitdrehen der kamera in fahrtrichtung ein- und ausgeschaltet
- „r" (wenn man im menü ist) löschen der aktuellen bestzeit
- „esc" - rückkehr ins menü, falls man gerade in einem spiel ist bzw. verlassen des spiels falls man im menü ist
- F1 – F9 (experimentieren mit opengl – wird unten noch genauer beschrieben)

wir haben das programm mit visual studio 2003 programmiert und das gesamte kompilierbare projekt ist im ordner /src.

ziel des spieles:
ein rennen geht über drei runden und es gilt den „ghost", der die schnellste bislang gefahrene zeit fährt, zu schlagen. das jeweils schnellste rennen wird also gespeichert und bildet den neuen gegner. (man kann nicht mit dem „ghost" kollidieren) wenn man im menü ist kann man mit der taste „r" die aktuelle gespeicherte bestzeit wieder löschen und das spiel sozusagen „resetten". das erste darauf folgende spiel ist dann ohne gegner.

was ist gemacht worden:
- model loader für milshape3d objekte (nichttriviale objekte)
- steuerung durch pfeiltasten
- skybox und boden (texturierung)
- kameraposition (kreisen um das objekt und dem objekt folgen)
- strecke (durch polygone)
- collision detection
- beleuchtung
- gegner
- menü + spielablauf (schrift (transparenz), rundenanzeige, startcountdown)
- spezialeffekte (bewegtes wasser + bewegte wolken (transparenz))

- experimentieren mit opengl (teilweise, siehe unten)

beschreibung:

die datei spiel.cpp enthält den hauptteil des programms, wie die erzeugung des fensters mit glut, die darstellung der frames in der display() methode, die steuerung, kamera, collision, gegner,...

model loader:
der model loader für .ms3d dateien (model.cpp und milkshapemodel.cpp) wurde im wesentlichen mit der anleitung aus http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=31 bzw. http://www.codeworx.org/opengl_tut31.php erstellt. das hovercraft, der gegner sowie herumstehende objekte sind mit milkshape gemodelt. in spiel.cpp werden in der methode modelle_laden() „MilkshapeModel" objekte erzeugt und zur darstellung in display() model->draw() ausgeführt. der boden, die skybox, und die strecke sind nicht mit models gemacht.

steuerung:
die abfrage der pfeiltasten erfolgt, wie auf der cg2 seite beschrieben, durch boolean variablen und eine myIdleFunc() methode. eine variable „speed" steuert die aktuelle geschwindigkeit des objektes und wird bei längerem drücken der „pfeil-oben" taste durch einen timer erhöht. lässt man die tasten los bleibt das objekt nicht sofort stehen, der „speed" wird langsam (auch mithilfe eines timers) erniedrigt.
das drücken der linken und rechten pfeiltasten ändert den richtungswinkel.
mit diesem richtungswinkel und der variable „speed" wird in jedem durchlauf die neue position berechnet. (variablen posX und posZ)
diese neue position hängt zusätzlich von der framerate ab, damit das objekt bei computern mit verschiedener rechengeschwindigkeit immer gleich schnell fährt.

skybox und boden:
die skybox besteht aus 8 rechtecken in einer display list, die um das hovercraft plaziert sind und immer den gleichen abstand zum objekt haben. der oktaeder fährt also immer mit der aktuellen position mit, man kann ihn also nie erreichen.
der boden (nicht die strecke) besteht aus großen quadraten (auch in einer display list) die texturiert werden. zum texturieren wird ein „GLuint loadtex()" objekt erzeugt wobei eine bmp datei eingelesen wird und in display wird mit diesem objekt „gebindet". (zB glBindTexture(GL_TEXTURE_2D, floortex);)

kameraposition:
die position der kamera wird mit gluLookAt() bestimmt, wobei der betrachterpunkt und der dort_schau_ich_hin punkt übergeben werden. der dort_schau_ich_hin punkt ist die position des objektes (hovercraft) und der betrachterpunkt ergibt sich aus dem abstand zum objekt und einem winkel (cam_winkel), der durch drücken von „a" und „s" gesteuert wird.
ist das folgen der kamera aktiviert („d") fährt die kamera immer einem speziellen winkel (zB hinter dem hovercraft oder seitlich; das kann durch „a" und „s" während „d" aktiviert ist gesteuert werden) nach. dies geschieht aber langsamer, als sich das objekt selbst dreht, da die kamera sonst immer sofort mitspringen würde und das nicht so gut aussieht. außerdem dreht sich die kamera nur, wenn man das objekt um mehr als 30 grad in eine richtung dreht, da sonst abermals andauernd hin und her gesprungen wird.

strecke:
die strecke wird durch 2 polygone (innen und aussen) beschrieben. zwei objekte punkt() bilden ein objekt Edge() und diese „edges" werden in einem Vector gespeichert. anhand dieser polygone werden die streckenwände (abermals in einer display list) gezeichnet. Die Texturkoordinaten für den Boden innerhalb der Strecke entsprechen den skalierten (auf 1mal1) Punktkoordinaten.

collision detection:
wir haben verschiedene ansätze für die kollisionserkennung versucht („hearn & baker" lieferte uns letztendlich die entscheidenden informationenen), und sind bei folgender methode verblieben. zunächst werden pro durchlauf nur die kanten berücksichtigt, die das objekt auf der x oder z achse schneiden. es wird eine translation mit diesen „active" kanten gemacht, so, dass der punkt mit dem kleineren wert auf der x-achse auf den koordinatenursprung verschoben wird. danach wird eine rotation der kante gemacht, genauso, dass sie auf der z achse liegt. diese transformationen machen wir natürlich auch mit dem aktuellen punkt des objektes, und schaun dann, wo der punkt auf unserem neuen relativen koordinatensystem liegt. zusätzlich berechnen wir noch den nächsten punkt in diesem koordinatensystem. hat nun ein punkt einen negativen wert und der darauffolgende einen positiven (oder umgekehrt) ist eine collision erkannt und die geschwindigkeit wird auf 0 gesetzt. wir haben versucht, dass das gefährt, abhängig vom einfallswinkel, wieder abprallt, jedoch hat es dann nicht so richtig funktioniert und war auch nicht so gut spielbar. alles in allem ein sehr langer und nervenaufreibender prozess.

beleuchtung:
alle eingebundenen milkshape objekte sind direktional beleuchtet, dh eine „unendlich" weit entfernte lichtquelle mit paralellen lichtstrahlen fällt auf

die objekte ein. die normalvektoren sind im .ms3d Format schon mitgespeichert und werden für jeden knoten mit glNormal3fv() im model loader (model.cpp) gesetzt. ein normalvektor ist durch seine umliegenden flächen gewichtet, durch glShadeModel(GL_SMOOTH) erscheinen objekte, deren poligone (dreiecke) eine runde fläche annähern auch wirklich rund. der boden, die strecke und die skybox wurden nicht beleuchtet, da es nicht wirklich sinnvoll ist.

gegner:
der gegner fährt immer die beste bislang gefahrene zeit. um dies zu realisieren müssen wir ein vom spieler gefahrenes rennen aufzeichnen können. dies geschieht durch das speichern jeder position des spielers in einem vektor (das spieler-objekt fährt, in dem es, immer von der alten position ausgehend, mit dem „speed" und dem richtungs-winkel, die nächste position ausrechnet).
wir merken uns also in jedem durchlauf die aktuellen koordinaten, sowie den winkel, den das objekt im welt-koordinatensystem hat. (dies führte allerdings später beim „abspielen" des gegners zu schlechten ergebnissen; der gegner hat stark geruckelt. daher speichern wir 4mal so viele „feinere" koordinaten ab als eigentlich pro durchlauf berechnet werden. dies führte zu wesentlich besseren ergebnissen. wir kamen nach langen überlegungen drauf das dies wahrscheinlich wegen dem nyquist-shannon-abtasttheorem so war)
nach dem startcountdown beginnt also die aufzeichnung und die zeit beginnt zu laufen. wenn man drei runden absolviert hat und die ziellinie erreicht wird die zeit gestoppt, ist man schneller als die alte zeit werden die koordinaten, die winkel, sowie die neue bestzeit (in der ersten zeile), in der datei gegner.txt im ordner data gespeichert (methode datei_aufnahme_speichern ()).

genauso wie bei jedem rennen aufgezeichnet wird, wird die alte bestzeit (sofern schon eine existiert) abgespielt. dazu werden die koordinaten und die winkel aus gegner.txt wieder in vektoren geladen (methode datei_gegner_laden ()). die erste zeile in gegner.txt ist die, zu schlagende, bestzeit.
es wird nun in jedem durchlauf aus der bestzeit, und der zeit, die im neuen rennen schon abgefahren ist der anteil berechnet, der schon abgefehren ist also:

anteil = bereits_abgefahrene_zeit / bestzeit

der anteil ist am start nahe null, und am ende der gespeicherten bestzeit eins. dieser anteil wird nun auf die anzahl der (koordinaten- und winkel-) einträge (=anzahl_einträge_in_vektor) skaliert, und dadurch ergibt sich für jeden durchlauf die neue position des gegners. dh:

eintrag_der_aktuellen_position = anzahl_einträge_in_vektor / anteil

die einbindung des gegners war, neben der kollision, sicher der 2. sehr mühsame prozess.

menü + spielablauf:
nachdem das aufzeichnen und abspielen des gegners nun möglich war, war es an der zeit dem spiel einen zeitlichen ablauf zu geben. dies geschah im wesentlichen durch einige flags, die das spiel in verschiedene zustände bringt, also menü, startsequenz, rennen_gestartet, nach dem zieleinlauf,...
dafür war es nötig schrift einzubauen. um schrift darzustellen wird ein bmp eingelesen (mit alpha kanal um transparenz zu erzeugen) und für jeden buchstaben eine display list mit den entsprechenden texturkoordinaten erzeugt. wenn nun ein wort dargestellt werden soll werden einfach die dezimalen ascii werte der jeweiligen buchstaben mit glCallLists() übergeben und das wort wird (nicht perspektivisch, sondern orthogonal und ohne depth testing) dargestellt. die methode, die aufgerufen wird um etwas zu schreiben ist glPrint(„text").
die rundenanzeige funktioniert, in dem flags gesetzt werden, falls das objekt bestimmte abschnitte überfährt (wie checkpoints)
nach 3 runden, sobald man die ziellinie erreicht kommt man wieder in den zustand menü, wobei, das spiel abgespeichert wird, wenn man die bestzeit geschlagen hat.

spezialeffekte:
die informationen zu beiden spezialeffekten (bewegtes wasser und bewegte wolken) sind aus:
http://www.opengl.org//resources/code/samples/sig99/
die methode, die das bewegte wasser erzeugt ist draw_mesh() dabei werden dreiecke mithilfe eines triangle strips erzeugt und deren koordinaten bei jedem durchlauf anhand einer sinus-schwingung verändert. dadurch ergibt sich eine bewegte wasseroberfläche. diese oberfläche wurde im spiel als eine art wassergraben (mit rampen) eingebaut, wobei sich die (hovercraft) objekte, falls sie über die rampen fahren erstmals in y-richtung bewegen. den effekt, also die bewegten dreiecke, sieht man sehr gut im GL_LINE polygon mode (mit F3 ein und ausschaltbar)
die bewegten wolken sind direkt in die display() methode eingebaut und funktionieren einfach mit transparenz und bewegten texturkoordinaten auf einem quadrat. im GL_TEXTURE matrixmode kann man die texturkoordinaten einfach mit glTranslate verschieben, was hier auch in jedem durchlauf geschieht. die transparenz kann man mit F9 aus und einschalten.
die geschwindigkeit beider effekte wird durch die framerate gesteuert, was ausreichen sollte, jedoch muss man hierbei erwähnen, dass diese methode nicht bei allen rechnern (vor allem bei schnellen rechnern wird die wellenbewegung des wassers sehr schnell) vollkommen zufriedenstellend ist. das ergebnis wäre sicher besser, wenn man es über einen timer steuern würde.

experimentieren mit opengl:
dieser punkt wurde wie folgt implementiert:

F1: hilfe gibt es keine, sollte auch nicht notwendig sein
F2: die framerate wird links unten ein- bzw. ausgeblendet
F3: wireframe modus wird aus- eingeschaltet
F4: texturqualität (nearest pixel vs. linear interpolieren) ein/aus. der qualitätsunterschied ist am boden sehr gut erkennbar.
F5: mipmapping ein/aus der effekt ist abermals am boden, in horizontnähe, am besten erkennbar.
F6: VBOs nicht implementiert - siehe unten
F7: ein und ausblenden der display lists nicht sinnvoll einsetzbar – siehe unten
F8: view frustum culling nicht implementiert – siehe unten
F9: transparenz (bei schrift und himmel) wird ein- ausgeblendet

der punkt, der leider aus zeitlichen gründen nicht wirklich berücksicktigt wurde ist der preformance vergleich, also vertex buffer arrays vs. vertex arrays vs. immediate, sowie frustum culling. alle objekte, bis auf die mittels model loader erstellten objekte, wurden mit display lists erzeugt, jedoch wäre es wahrscheinlich auch nicht der sinn der übung gewesen, diese objekte einfach auszublenden und nicht zu zeichnen.
hierbei sei vielleicht noch einmal erwähnt, dass wir ohnehin schon sehr viel zeit in die übung investiert haben und wir es für sinnvoller hielten, alle diejenigen punkte zu erfüllen, die man im spiel auch unmittelbar sehen kann. ansonsten hat uns die übung sehr viel spaß gemacht und wir konnten sehr viel dabei lernen.