

CS 451 Advanced Computer Graphics, Spring 2009

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

This course covers advanced topics in computer graphics. This year the course will cover 3D video game design. Using a commercial 3D video game engine, students will divide into groups to design and implement a playable video game with at least one complete level. In addition to learning to script the game engine, students will have the opportunity to modify the game engine, develop 3D content, and animate 3D skeletons. The course will also cover selected topics in advanced graphics related to game design.

Prerequisites: CS 351 or permission of instructor

This material is copyrighted. Individuals are free to use this material for their own educational, non-commercial purposes. Distribution or copying of this material for commercial or for-profit purposes without the prior written consent of the copyright owner is a violation of copyright and subject to fines and penalties.

1 Course Arc

- **Game Genres/Analysis:** Look at existing games, traditional game classes, and the best games in each category. Identify what you like about games.
- **Game Planning:** The process of building a game from design to implementation, including: roles of game designers, concept visualization, content creation, and overall system design.
- **Game Rules:** What rules are essential to gameplay? What is the purpose of rules? How do we decide what the rules ought to be? How do we relax the rules without creating unreasonable demands on content and scripting?
- **Game Interfaces:** What makes a good interface? How do the rules, content, and overall design interact with the physical interface? How does an interface impact gameplay?
- **Game Content:** How to create content for a game. We will look at the process of designing 3D models, texturing/skinning the models, animating the models, and generating the necessary structures to import them into a game engine. We'll also go over basic forward and inverse kinematics and how to create realistic motions.
- **Game Scripting:** The skills and knowledge required to implement the game using a game engine. We'll be using the Torque 3D game engine (or the 2D game engine if you want to do a 2D game). Our focus will be on TorqueScript, which is similar to C/C++ in syntax and form, but is interpreted.
- **Modeling and Simulation:** Part of designing simulation games is modeling systems such as economic, political and physical systems. How do we develop such systems and make them appear realistic to the user?
- **AI:** A competitive opponent is a critical part of gameplay for single, and even multi-user systems. We'll look at a number of techniques we can use to give the computer the intelligence to oppose the player. At an even more basic level, we'll go over how to make NPCs and bad guys act appropriately given the current context.
- **Audio:** Audio is an important piece of game interfaces, and the Torque game engine supports the idea of 3D sound sources within the engine—sound sources that are localized to particular objects or characters in the world. We'll look at audio as part of the interface, the function of different kinds of audio tracks, creating interesting noises, and how to integrate them into a game.
- **Usability:** How do we measure the usability of a game or a game interface? We'll look at how to design a usability study and how we can use them to get feedback for improving game design.
- **Multiplayer Games:** Many of the major games out today are multi-player games, either pitting human against human or requiring people to work together on quests. We'll look at networked or multi-player games both in general and the specifics of the Torque game engine.
- **Special Effects:** Special effects are a big part of modern games, from explosions to natural motion and physical simulations. We'll take a look at how some effects can be implemented in a game engine and how vertex or pixel shaders can provide additional capabilities.
- **Collisions:** Collisions are a critical piece of game engine design, and likely constitute one of the major time commitments for the processor. We'll look at different ways of detecting collisions and how they are integrated with the overall form and function of the game engine.
- **Advanced Graphics Topics:** We'll look at other advanced graphics topics with any remaining time.

2 Intellectual Property

The importance of intellectual property in video game design cannot be understated. The video game industry is a multi-billion dollar industry and relies almost completely on intellectual property rules. The flow of information is incredibly fast, and there are so many people looking at games that you can be sure any violation of copyright will be quickly identified. You want to make sure that anything you put in your game is original work.

2.1 Copyright

Copyright is the most important aspect of intellectual property with regard to video games, although trademarks are also relevant for names and phrases.

Copyright applies automatically to original works of art, writing, and code. Copyright also covers the right to create derivative works, so tread carefully when using ideas from novels, books, or movies. You do not need to do anything for your work to be copyrighted. However, you may want to notate and identify dates of creation and who created the work in case there is ever a question.

All material from all video games ever created is copyrighted. Some material may have been explicitly released into the public domain, but none of the copyrights have expired. Copyrights last for the life of the author plus 70 years, or for 95 years from publication/120 years from creation in the case of works for hire (whichever is shorter). Because of an act of Congress in 1998 that extended copyright by 20 years, no materials will enter the public domain until 2018.

2.2 Copyright: Fair Use

In video games there is no fair use unless you are creating your video game for personal reasons with no intention of sale or profit. Fair use depends upon four factors, most of which work against using copyrighted material in video games:

1. The purpose and character of the use, including whether such use is of a commercial nature or is for nonprofit educational purposes
2. The nature of the copyrighted work - highly personal works get more protection
3. The amount and substantiality of the portion used in relation to the copyrighted work as a whole
4. The effect of the use upon the potential market for or value of the copyrighted work

The bottom line is that you want to take your own pictures, avoid using trademarked items without permission, and don't just grab stuff off the internet without thinking about what you're doing.

I'm working under the assumption that Colby makes no claim to student intellectual property generated as part of a course. Therefore, your group needs to make a decision about how IP will be handled.

As the development of a video game is both a group activity and an activity with the potential for profit, it is important for each group to develop an intellectual property agreement that treats fairly all members of your group as well as any other person who provides IP used in the game.

Assignment 1: Develop an IP agreement for your group

3 Game Genres and Analysis

What is a game? Crawford has an interesting diagram in his *On Game Design* book that uses a tree of concepts to separate games from other forms of activity. Games fall into the broad category of creative expression. Then he asks five questions about characteristics of the activity.

For Money?	Interactive?	Goals?	Competitor?	Attacks?	Category
No	No	No	No	No	Art
Yes	No	No	No	No	Entertainment
Yes	Yes	No	No	No	Toys
Yes	Yes	Yes	No	No	Puzzles
Yes	Yes	Yes	Yes	No	Competitions
Yes	Yes	Yes	Yes	Yes	Games

Many programs we call computer games are not actually games, according to Crawford. SimCity, for example, is a Toy in some modes, but would probably fall into the category of a Puzzle. Many computer games are race games where competitors cannot impede one another. Small modifications to competitions, however, can turn them into games, such as allowing race cars to bump into one another.

Creating a taxonomy of game types is not a trivial task. In some ways, the genres reflect a lead successful game that offered something new that was then copied and modified by successive games within the genre. Games can be divided by a number of different variables, and a single game might contain multiple values of a single variable, especially if the game involves a series of different kinds of challenges.

- Input device: keyboard, mouse, joystick, gamepad, specialized input device, wireless device
- Number of players: single player, multi-player, massively multi-player
- Iconography: science-fiction, fantasy, card games, realism, abstract, ...
- Speed of play: turn-based, continuous action, simulation with time control
- Number of actively controlled units: one, small group, large units and formations
- Source of challenge: physical speed, reflexes, problem solving, multi-tasking, strategy
- Level of Control: player controls all action (e.g. card games), player only indirectly controls action (e.g. The Sims).
- Point of View: first-person, third-person, overhead
- Purpose: education, mental stimulation, enjoyment, training, therapy
- Richness of environment and environmental interactions: simple, linear, free-form
- Goal: achieve a state of harmony, kill the bad guys, self-improvement, beat the competition, education

There are a number of different taxonomies of video games, and new games have blurred the line between different types. For example, Wolf (Chapter 6, "Genre and the Video Game" in *The Medium of the Video Game*, Wolf ed., 2002), proposes a more specific, horizontal taxonomy, where many (if not all) games belong in more than one class.

Wolf Proposes the following categories:

Category	Description	Example
Abstract	Non-representational graphics, not much narrative	<i>Tetris</i>
Adaptation	Adaptation of a game from another medium or activity	<i>Price is Right</i>
Adventure	Complex interactions with a rich environment	<i>Raiders of the Lost Ark</i>
Artificial Life	Simulation of creatures requiring input from the user to achieve some goal state	<i>The Sims</i>
Board Games	Video game adaptations of board games	<i>Chess</i>
Capturing	Catch stuff that runs away	<i>Gopher</i>
Card Games	Video game adaptations of card games	<i>Bridge</i>
Catching	Catching stuff that may move, but does not actively run away	<i>Big Bird's Egg Catch</i>
Chase	Pairs with Catching, Capturing, Driving, Escape, Flying, Racing	
Collecting	Collecting things that don't move	<i>Pac-man</i>
Combat	Equally matched projectile-shooting entities	<i>Battlezone</i>
Demo	Game demonstration, usually not fully-featured	
Diagnostic	Diagnostic programs (not really games)	
Dodging	Primary objective is avoiding projectiles or moving objects	<i>Frogger</i>
Driving	Games based primarily on driving skills	<i>Pole Position</i>
Educational	Games designed to teach	<i>Mario Teaches Typing</i>
Escape	Objective is escape from pursuers or a maze	<i>Ms. Pac-Man</i>
Fighting	Hand-to-hand combat	<i>Mortal Kombat</i>
Flying	Games that involve flying skills	<i>Microsoft Flight Simulator</i>
Gambling	Adaptations of gambling games	<i>Blackjack</i>
Interactive Movie	Branching video clips	<i>Star Trek Borg</i>
Management Simulation	Resource management games	<i>SimCity</i>
Maze	Objective is successful navigation of a maze	<i>Lode Runner</i>
Obstacle Course	Traversing a set of obstacles	<i>Boot Camp</i>
Pencil-and-Paper Games	Games usually played on paper	<i>Hangman</i>
Pinball	Self-explanatory	<i>Pinball Wizard</i>
Platform	Moving through a series of levels and avoiding obstacles	<i>Donkey Kong</i>
Programming Games	Player writes short programs to control agents	<i>CoreWar</i>
Puzzle	Primary objective is to solve a problem or a riddle	<i>Myst</i>
Quiz	Tests player's ability to answer questions	<i>Trivial Pursuit</i>
Racing	Primary objective is to win a race	<i>Pole Position</i>
Rhythm and Dance	Objective is to keep time with and respond to music	<i>Guitar Hero</i>
Role-Playing	Player takes on the persona of the character	<i>Diablo</i>
Shoot 'Em Up	Shooting lots of opponents or objects (one-sided battle)	<i>Space Invaders</i>
Simulation	Meta-category for Management and Training Simulation	
Sports	Adaptations of sports	<i>Madden NFL 2007</i>
Strategy	Primary skill required for success is strategy; often turn-based games	<i>Spaceward Ho!</i>
Table-Top Games	Adaptations of table-top games	<i>Pong</i>
Target	Primary objective is hitting a target	<i>Shooting Gallery</i>
Text Adventure	Adventure/role-playing game with a text-based interface	<i>Zork</i>
Training Simulation	Simulation of realistic situations	<i>Flight simulators</i>
Utility	Programs with functionality beyond games	

Obviously, one could develop alternative taxonomies, and perhaps a hierarchical taxonomy that divides along a sequence of variables might offer more interesting insight. After all, the purpose of a taxonomy is to create an abstract representation of a complex set of data in order to permit meta-reasoning and understanding of complexity.

3.1 What makes a good game?

We can divide games into genres along various different axes, but just because a game is in a particular genre does not mean it is fun to play (even if you generally like the genre). You might really enjoy SimCity, play SimTower every once in a while, but SimFarm is pretty boring (IMO).

What are the factors that make gameplay good, or fun?

Three basic concepts to consider are identified by Salen and Zimmerman:

1. Meaningful play: actions of the player have meaning by how they affect the outcome of the game.
2. Discernable effects: the effects of actions (or inaction) must be discernable to the user.
3. Integrated effects: player actions should have not only an immediate effect, but a long-term effect on the game outcome.

Here's my list of generic things that make for good gameplay:

- It has to be challenging (there has to be a reasonable chance of failure).
- You should be able to improve with play.
- You should always feel like you can win if you try harder, practice, and/or make the right decisions.
- There should be intermediate rewards.
- The game should be consistent within itself.
- It should feel like your opponents are playing fair.
- The game should offer complete functionality within the format of the game world.
- The interface should be usable with a small amount of practice.
- The interface should make all information required for successful gameplay easily visible to the user.
- All actions by the user should impact the outcome and larger context of the game.

There are also genre-specific issues, which is where things like ambience, visual effects, expected capabilities, and player interaction probably fit.

One goal of video games, which is enabled by the visual aspect of the medium, is putting the player in a zone where the game is their perceptual world. What are the characteristics of a game that enable this effect?

- Quality/smoothness of the interface: does it fit naturally with the game actions?
- Time: a feeling of time pressure by the player contributes to the zone (whether or not it exists).
- Engagement: the game process has to be interesting and achievement desirable.

What are some questions we can use to help us evaluate games?

Here's my starting list of generic questions:

- Rate the level of challenge the game presented.
- Rate how much you cared about doing well at the game.
- Rate degree to which the interface provided relevant information.
- Rate the difficulty of the interface.
- ...

Develop a standard set of questions for evaluating games you play this semester. Fill one out any time you play a game for more than an hour (but only once form per game).

4 Game Planning and Design

Realistic goals (Kanade rule): If you think you can finish a project in X units of time, double X and increase the units of time by an order of magnitude. *For a 4-month project, pick something you think you could finish in 2 weeks if you had nothing else to do.*

Part of setting realistic goals is limiting the space of possibilities. In any design project, it is always possible to make the design better or add more features if you have more time. But time is not an infinite quantity. Limiting your game design to fit within certain constraints can also encourage more creativity. It can be difficult to be creative in the absence of constraints. Reasonable constraints force us to be creative.

The following are some game parameters you may want to constraint to help limit the complexity of the design.

- Number of players: limit to a maximum number.
- Expected time to complete a level of the game: quick game or long?
- Game purpose: pick something that interests you.
- Game skill: what does the player have to learn to be good at?
- Genre: will your game conform to a specific genre type?
- Narrative quality: is the game a linear story, branching, or free-form?
- Number of levels: one, N , or infinite?

Iterative design concept

Create (physically) a rough interactive prototype as quickly as possible with people playing people. Make the game out of paper and soda straws if you have to, but start with an initial set of rules and players and see if the ideas work.

Salen and Zimmerman Rule: 20% of the way into the game design, have a prototype (3 weeks)

Game Design Process (Crawford)

- Goal and topic development: figure out what you want your game to do. Think about the purpose of your game, and make it something you care about. Then select a topic that is appropriate.
- Research and preparation: do your homework on your topic so you get both the big picture and the details right.
- Design phase: develop the overall game concept.
 - User interface design: how does the user interact with the game and execute actions. How does the computer provide information to the user?
 - Game structure: what are the key elements of the game? Where does strategy or decision-making come in? What are the options of the player with respect to the key element? The game must have a balance between permitting expressiveness and limiting choices to permit smooth continuous action.
 - Program structure: what are the key structures required by the interface and the game structure? How do they communicate? How is the data organized?

- Evaluation of the Design (go-ahead decision is here): play the game as a prototype version on pencil and paper, or with little figures. Consider stability and balance. Does the game achieve the goals you set out at the beginning?
- Pre-programming phase: documentation of the game design.
- Programming phase: implementation of the game design, with lots of iterations.
- Playtesting phase: testing of the game implementation, with lots of iterations.
- Post-mortem: how is your game?

4.1 Developer Roles

Producer: The producer is the overall organizer and manager of the project. This person does not need any special expertise, but some knowledge of all the areas of game design is important. The producer is the one who watches the calendar and provides motivation where needed, and assistance when required, perhaps by diverting resources from one part of the project to another. In a real game development situation, the producer also handles most of the outside contact and money issues.

Designer: The game designer develops the rules, concept, goals, interface, and the various and sundry other pieces of a game. The designer is good at generating, testing, and weeding out ideas. Most members of the game development team play some role as a game designer. The lead designer is responsible for integration of the ideas into a playable game.

Programmer: The game programmer puts the design concepts into practice. The challenge is figuring out how to implement the ideas put forward by the design team.

Visual Artist: The visual artist develops the visual content for the game. Most game engines use abstraction to separate the specifics of the visual art from the game programming so that updating visual art is a simple process.

Audio Artist: The audio artist develops the audio content for the game.

Quality Assurance Specialist: The QA person is in charge of testing and evaluating the game as it develops. Most members of the team will take part in QA, but the lead QA person is responsible for ensuring that all aspects of the game have been tested.

4.2 Planning Tools

Critical Path Management

Critical path management [CPM] is a technique for identifying bottlenecks in the development process.

- List each task in the design process, dividing the process into small, but natural chunks of work. Each task should have a clear outcome or product.
- For each task, estimate how long the task will take, list the inputs to the task and list the output of the task.
- Arrange the tasks along a timeline, showing the relationships between inputs and outputs. Start at the end and work backwards. Place tasks in parallel wherever possible.

CPM can help you identify critical tasks along the development path. Often, several parallel streams will converge, and work cannot continue until a single task is complete. Dividing tasks into pieces also helps you estimate the amount of time a task or project will take.

Versioning Control System

Set up something (e.g. cvs) so that you have a versioning system working that manages the issues of multiple people working on the same project. It is quite reasonable to make a CVS repository for all of your Torque scripts.

4.3 Integration

Integration always takes longer than you think. If you have people working on parallel tracks, plan for at least 20-40% of your overall time to be integration of the pieces. Testing is a necessary part of integration because there will be problems in code, etc., that don't show up until things are integrated. One of the biggest challenges with Torque is maintaining name spaces properly. For example, if two design teams use the same name for different player, button or GUI elements, that will cause problems when the system is integrated.

5 Designing for an Audience

Author's note: need some more general guidelines about designing to an audience here. Basically, know your audience and do some research to figure out what they like.

Question: who is your audience?

If we do not regularly state that a percentage of our audience is expected to be female, we assume we are designing for males.

– Sheri Graner Ray, *Gender Inclusive Game Design: Expanding the Market*, 2004.

Most of the myths of designing neutral or equitable gender games have been shown to be false.

Myth 1 Males won't purchase games with female lead protagonists.

In 1989, one of the first game companies founded by a woman was hesitant to have a lead female character, because they didn't want to lose their male audience. However, the first game they sold with female protagonists—*King's Quest IV*—had even better sales.

Tomb Raider likewise shattered the myth. Sony enjoyed huge sales and made the game part of the playstation rollout. Sega and Nintendo had refused to build similar female lead characters, believing it would threaten their main market: young men.

Myth 2 Females don't enjoy video games as much as males.

In 1999, *The Sims* completely dismissed the myth that women don't play video games

- *The Sims* became the biggest selling title of all time soon after its release
- It is estimated that over 60% of Sims players are women
- *The Sims* was subsequently beaten by *The Sims II* and *The Urbz: Sims in the City*

Myth 3 Females are less skilled at computer usages than males

Research has shown this to be false.

- Equal exposure to computer games decreases pre-existing gender differences (Greenfield, 1996)
- Given equal access to computers, there are no gender differences in programming ability (Linn, 1985).
- A generally more positive attitude and more experience with computers among males does not translate into higher performance in computer courses (Woodrow, 1993).

Conclusion: There is a big market out there that includes men and women in equal measure.

5.1 Gender considerations in design

Research shows there are gender differences in the way games are played and designed.

In one study (Van Eck, 2006), when teams of all boys, all girls, or mixed genders designed games, both tended to make adventure games centered around exploration.

- Boys tended to make conflict the central element of the game
- Girls tended to avoid direct conflict as a solution and required players to use other options

In the same study, when boys and girls played the same game, *Sim Safari*, they enjoyed it equally well, but they used different parts of the game.

- Girls designed high-end, detailed dwellings
- Boys designed swamps, crocodiles, and jaguars.

A second study (Hartmann, 2006) surveyed German women to determine their impression of a set of fake video games designed to evaluate three factors: social interaction, violence, and physical stereotypes. The average responses rated opportunities for social interaction the most important factor, followed by a non-sexualized role for the female protagonist, and then non-aggressive content. Almost half (44% of respondents) reversed at least one of the factors with respect to the average order, although the social interaction aspect was generally not reversed with respect to the other two.

In the same study, male and female respondents differed in their intensity of gameplay based on competitiveness. In non-competitive genres, no difference was observed.

Why do we care?

Historical facts:

- Up through 1991 92% of arcade games contained no female roles whatsoever; 2% had active female roles.
- Less than 10% of the audience for traditional PC games is female
- Less than 15% of Nintendo's user base was female before the wii
- Less than 20% of the audience for traditional online titles are female
- 70% of casual, online gamers are female (44% overall online gamers, (ESA, 2005))
- 52% of internet users are female
- The video game industry had over \$10.5 billion in U.S. sales in 2005 and is expected to reach almost \$50 billion in worldwide sales by 2011 (excludes hardware sales)
- As of 2005, in the US, 43% of all video game players were female (ESA, 2005))

There is clearly a place, and money to be made, for games that are gender neutral.

But building games with gender in mind does not mean building around stereotypes. Initial attempts at building games for women failed miserably.

The industry took an entire market of women and defined it as a genre of "fashion, shopping, and makeup games for girls ages 6-10."

– Sheri Graner Ray, Gender Inclusive Game Design: Expanding the Market, 2004.

One of the exceptions was Mattel's *Interactive Barbie*, which was more of an interactive design space and in many ways an accessory for physical Barbie dolls.

Gender considerations in design do not mean stereotyping the audience. Instead, it has more to do with balanced design and subtle genre considerations. One interesting observation is that gender stereotypes, in some ways, more strongly affect men than women. The relevance to game design is that to design games women will find enjoyable, we really don't have to go out of our way to actively design elements of the game to appeal to women; **we just have to avoid designing our game solely to fit male stereotypes.**

Some of the guidelines proposed by Shei Graner Ray, (an academic, and founder of a game company focused on games for women), include the following (which are clearly some broad generalizations):

- Learning styles: Males tend to more explorative and risk-taking. Females want to know how something works first and tend to model or imitate as part of learning.
 - Most tutorials in current games are explorative in nature.
 - Design: generate tutorials with imitative models as well as explorative models
 - Educational software provides many examples
 - The learning style difference is not necessarily limited to gender, or explorative v. imitative. As a designer, consider visual, audio, and haptic learning differences when building tutorials.
 - **However: the most successful cross-gender games are exploratory in nature, so there is a separation between learning a game and playing a game.** (Van Eck, 2006)
- Price of failure: Males expect punishment for error, females forgiveness (very broad generalization). Consider the context of an RPG game where when you die you lose something irretrievably.
 - Design: design victory conditions such that failure to meet those conditions does not result in irretrievable loss.
- Avatars: since they often represent heroic characters, avatars normally exhibit exaggerated physical characteristics. Female avatars are often given exaggerated sexual characteristics, while male avatars are given exaggerated size and muscles. Whether these constitute sexual characteristics is arguable, but certain male sexual characteristics are generally not exaggerated.
 - Consider that culturally, it is more difficult for males to be “sissies” than females to be tomboys.
 - Design female avatars based on female athletes, not female porn stars (avoid hyper-sexualization)
 - Focus group test all avatars with female characters (don’t assume anything)
- Communication: Males and females tend to communicate differently, even in electronic media. In particular, there is gender preference for formality and rapport-building language.
 - Avoid industry or genre specific jargon in documentation, tutorials, and game scripts (e.g. WASD).
 - Avoid using content containing sexual humor and negative comments.
 - Be inclusive about including formality and rapport building language in commands
 - Real social interaction is interesting.
- Production environment: who you are defines, in part, what you design.
 - Avoid “he” in all documentation and tutorials. Do not assume your player is a male.
 - Include women in the design and testing process.
 - Clearly state who constitutes your intended audience. If your audience does not include females, you are cutting out half of your potential market share.

6 Rules

Rules are an essential part of gameplay, and games cannot exist without them. One of the characteristics of game rules is that they rarely have any impact outside of the game. This characteristic makes it possible to create fantastic, fanciful, annoying, whimsical, or difficult rules as part of a game, so long as the gameplay is challenging, engaging, and fun.

Nevertheless, rules have qualities that are essential to good gameplay (from Salen and Zimmerman).

- Rules limit player action
- Rules are explicit and unambiguous
- Rules are shared by all players
- Rules are fixed
- Rules are binding
- Rules are repeatable

There are also different kinds of rules, all of which are necessary to make a game enjoyable.

- Operational rules: the rules that define how to play the game from a purely functional point of view.
- Constitutive rules: underlying formal structures of the game. Understanding these structures can help you play the game. The structure also defines the search space for game play. These are the rules you need to understand in order to have a computer play the game.
- Implicit rules: rules that player's need to follow in order to make gameplay fun. For example, not taking too much time choosing a Scrabble word. In video game design, many facets of the interface—e.g. that the cursor will respond to mouse movement—are implicit rules.

6.1 Complexity and Emergence

Rules define a system. We can think of this system as defining a space of possible trajectories for the game. Each location in the space is called a state. A state represents a fixed-snapshot of a game that incorporates all of the information required for the game to continue.

tic-tac-toe: A simple game, like tic-tac-toe, has a space of possible trajectories that is definable and tractable to explore. One of the reasons it is generally played by young children is that they cannot yet encompass the space of game trajectories and outcomes within their minds, so the game has an exploratory, or surprise component to it. However, as we get older, the game loses that quality since we can encompass the total space of game trajectories.

chess: Chess, on the other hand, has a space of possible trajectories that is very large, and which no human has yet been able to encompass within their minds. While there are boundaries to the space we can encompass—e.g. initial tracks of play that result in quick victories—two good players will never play the same game twice and will explore very long trajectories of gameplay.

The complexity of a game is strongly related to its gameplay and level of enjoyment. The rules of a game define its complexity, but a complex game need not have complex rules.

Some ways we can add complexity to a game include:

- Random variables
- Prior player actions affecting current actions (feedback loops)
- Elements of the game being co-dependent upon one another (feedback loops)
- others?

In physical systems, complexity generally arises from one of two situations: non-linear systems, and feedback loops. In non-linear systems, small changes in initial conditions—e.g. small random variables—can cause large changes in outcomes. Likewise, systems with feedback loops can rapidly switch states, and/or respond to perturbations in interesting ways.

What is the difference between a linear and a nonlinear system?

- In a linear system, the response of the system is proportional to its inputs
- In a non-linear system, the response is generally more than linear (e.g. a power or an exponential), but can be sublinear (root or logarithm)
- Systems with feedback use not only the current input, but also prior state information to determine the next state

Linear System: consider a game where you can purchase technology. When designing the game, you have to decide what each step in technology means in terms of gameplay. A linear system would make each step in technology equivalent; if you can build one unit with technology level one, then you can build 2 units with technology level two, and so on.

Non-linear System: in the same game, a non-linear system would make each step in technology be a proportionally bigger (or smaller) step. If you can build 1 unit with level one, then you can build 4 units with level two, 9 with level three, and so on. Later technology steps cause much more significant changes in the game.

Combining linear or nonlinear systems with feedback also produces complexity, especially when the feedback crosses rules. For example, choosing to increase technology may inhibit or enhance the user's ability to upgrade other capabilities.

Emergence is a quality of a game that occurs when surprising things occur. The game of life, for example, has an emergent quality in that certain configurations of squares exhibit non-random behavior such as walking across a screen or maintaining a stable or semi-stable state.

Emergence often occurs when simple rules interact in unexpected ways. When designing a game, emergence is a nice quality of a game, and facilitates replayability. However, emergence is not always a good thing, and sometimes requires redesign of game rules to manage the effects.

6.2 Feedback Loops

Rules will inherently create feedback loops in a system.

Civilization: There is tremendous benefit in civilization to spending lots of resources on technology, if you can afford to do so. Increases in technology can enable your population to grow, your cities to thrive, and your military units to be so powerful that you only need a few to protect your civilization. It is a positive feedback loop, because a civilization with lots of technology can grow faster than a civilization with a lower technology, resulting in an increasing gap between them.

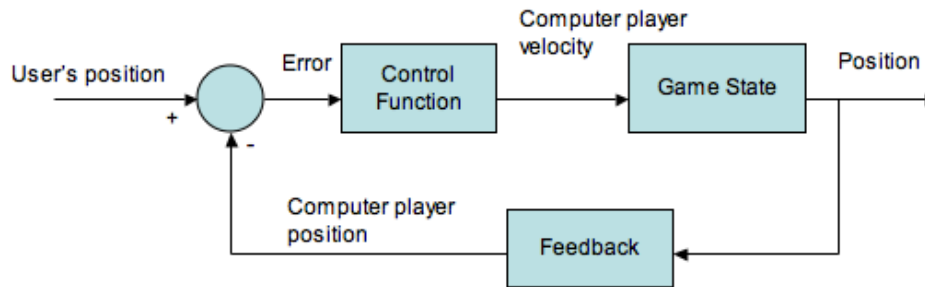


Figure 1: Feedback loop diagram for a racing game.

Such a **positive feedback loop** is important for gameplay, because it is one strategy a player can use to win the game. The whole point of the game is to balance and enhance the factors that improve your civilization.

However, as a player, if your situation is such that the game has an imbalance early on, then the game is effectively over because you have no way to survive against the superior technology. In that sense, the feedback loop can lead to early defeat due to small factors in your initial situation.

Negative feedback loops are also essential for gameplay. For example, in a racing game where some of the players are computer-controlled, the computer could enable its players to drive perfectly and the maximum possible acceleration. In that situation, it would be almost impossible for the human player to win (and certainly not for a beginning player), making the game frustrating.

A negative feedback loop helps the computer to make the game more interesting. When a player is far ahead of the nearest computer player, the computer can make its closest player play better (more like the human player), giving the person some real competition. Alternatively, when the human player is far behind, the computer can make several of its players slow down in order to keep the game interesting.

In both cases, the difference (or error) between the computer and the person is being reduced by the negative feedback loop, which makes the game more interesting.

From a generic point of view, the game can be represented as a loop with boxes. The goal is to keep the computer player close to the human player, which means we want to drive the difference between their positions—the error—towards zero.

As shown in figure 1, the user's position is input into the system. The difference between the computer player's position and the user's position is calculated and then fed to a control signal modifier that converts the error into a velocity signal for the computer vehicle.

The game state then gets updated, and we can measure the position of the computer player from the game state, which then gets fed back around through the feedback loop.

Feedback loops can form part of the operational rules, or part of the constitutive rules that manage the space of possible trajectories within the game. Since the computer's players are inherently linked to the actions of the human player through the feedback loop, the human will always have a different, and hopefully enjoyable, experience in each game, regardless of their skill level.

7 Game Engine

For a single-user, single platform game you could write a single monolithic game engine that contains a single copy of all game information. While it would be faster, there is no way it could support a multi-player game.

A server-client architecture is necessary for multi-player games, where the server maintains game state and the client handles user interactions. There are a number of possible arrangements of the server-client relationship on a single machine.

- Server and client both active on an isolated single machine: single-player arrangement.
- Server and client both active on a single machine, with server accepting external connections: multi-player game with one player on the server machine.
- Server active, client inactive: dedicated server arrangement, generally for massive multi-player games.
- Client active, server inactive: client program for connecting to an open server.

By using a client-server model for all game types, the flexibility of the game engine is significantly increased with a slight penalty in performance for single-user/single-machine games.

7.1 Game Engine Tasks

The engine is responsible for a large number of tasks within the game. The purpose is to relieve the game programmer of as much responsibility for low-level operations, letting you focus on the gameplay, interface, and the high-level response of the world to user actions.

Engine Responsibilities

- Networking & communication: the engine handles all of the low-level details of networking and communication between clients, or the client and server.
- Game state: the engine maintains the game state, which includes the current location, velocity, acceleration of all objects as well as the position of all static objects in the world. Game state also includes any state information associated with gameplay, NPCs, or AIs.
- Information/world consistency across clients: the engine is responsible for maintaining world consistency across clients so that any change in game state is propagated appropriately.
- Rendering: the engine is responsible for rendering the current views of each player.
- Collisions: the engine handles all collisions between all objects within the world, including keeping the player on the terrain (when appropriate).
- Scheduling and notification of events: the engine handles the ordering of all scheduling actions and generates all of the required callbacks for both client and server functions.
- Script interpreter: the engine handles interpreting script, although most scripts are compiled upon loading the mission.

Definition: A **callback** function is one that gets executed when a specified event occurs. Events that generate callbacks include: collisions, when an object is created, when an object is destroyed, or when a level is created or destroyed.

Division of labor: you need to divide the functionality of your game between server and client scripts. In short, client scripts should handle the interaction between a user's input and the user's player. Anything not directly affected by a user's actions should not be on the client side, including any AI or NPC control functions.

In a multi-player game, you don't want a player to be able to hack the client program and control world functions.

7.2 Torque3D Engine High-level Functioning

Finney, Chapter 5

When engine begins, it grabs a startup script, `main.cs` from a known location. For the kit we have, the location is `Torque SDK/example/main.cs`. The `main.cs` script loads other basic scripts and starts up the `main.cs` script in the game directory specified at the top of the `example/main.cs` script.

directory structure

top level: `Torque SDK/example/main.cs`

`common/{main.cs, prefs.cs, client, server, help, lighting, ui}`

Game directory/`{main.cs, prefs.cs, client/{ui}, data{interiors, missions, shapes, skies, sound, terrains, water}, server}`

There is a lot of common scripting provided with the engine that handles game startup and functions common to all games. These scripts are located in the `example/common` subdirectory.

Common code: basic functions and capabilities

- Sets up the video
- Creates the window for the game
- Sets a bunch of OpenGL parameters (which you can do within script)
- Opens a set of standard GUIs for the game window
- Executes a number of helper scripts that can be useful for creating a game
 - `metrics.cs` gives you access to performance metrics which can be overlaid on the screen
 - `messageBox.cs` gives you a message box with text or yes/no buttons
 - `screenshot.cs` gives you the ability to grab a movie or a single screen shot
 - `recordings.cs` lets you play a demo game recording
 - `cursor.cs` lets you control the cursor visibility and manage it appropriately
 - `help.cs` provides the framework for context help
 - `message.cs` handles passing messages (text or otherwise) between server and client
 - `mission.cs` defines the function `clientCmdMissionStart`, which gets called right before client is dropped into a game

- `missionDownload.cs` downloads datablocks and ghost objects, and lights the scene when a mission begins. There are three phases to the mission download, and these three phases synchronize with the same actions on the server.
- `actionMap.cs` handles something with interfaces, letting you map inputs from one object into another
- There are symmetric script files defined in the `common/server` subdirectory. For example, `missionDownload.cs` on the server side calls the function where your player actually gets created.
- Initializes the audio system

Within the `main.cs` file in your game directory, things happen in order.

Server gets initialized first:

- Common stuff
- `game.cs` in your game directory loads all of your scripts and defines a skeleton `onMissionLoaded` callback. `onMissionLoaded` is where you set up the level, create AIs, etc. The script `game.cs` also defines the function that creates a player within the game when a client connects. The creation of the player has to be server-side.

Client gets initialized after:

- Common stuff
- Loads all of the game specific scripts from the client subdirectory
 - GUIs
 - Scripts: here is where a lot of what you write will go
 - Key bindings
 - Preference settings for the game engine
 - Initial setup
- Client defines the `loadMyMission` function, which actually begins a game, because it is connected to a button in the interface.
 - Disconnect from a server (in case already connected)
 - Create a server using the specified mission file in the `data/missions` directory
 - Connect to the server
 - connect to an existing server for a multi-player game

Game Engine functioning

Game engine time moves in ticks. A tick is 32ms, which means the game engine processes events 30 times per second. Trying to force something to happen more often than 1 tick is not a good idea.

At each tick, the game engine is trying to identify collisions in the world. When a collision occurs between two objects, the game engine checks to see if the objects care about a collision. Objects can receive or send collisions.

- An object that does not receive or send collisions, but has a collision box, will affect things that run into it by not letting them pass through it, but it will not initiate a callback for itself or another object. Elements of the environment that are static, do not change, and do not affect the state of other objects fall in this category.
- An object that sends collision events, but does not receive, can initiate another object's onCollision callback. Static objects that do not change, but which may affect other objects, fall in this category. The bottom of a pit with sharp knives, for example, might trigger a callback on any object that fell into it. If two objects that can only send collisions collide, no callback is instantiated.
- An object that receives collisions but does not send them is generally something that is a temporary part of the environment, but does not change the state of the thing with which it collided. For example, an invisible door that disappears when touched does not affect the state of the player, but needs to know when it has been touched.
- An object that receives and sends collisions is an interactive part of the environment and its state is affected by collisions and it can affect other things with which it collides. A player or NPC, for example, needs to both send and receive collisions. Likewise, a health or power box needs to both know it has been touched (so it can disappear) and the player's state needs to be updated.

Only the server is notified of a collision, not the client. The server must notify the client of any change in state that needs to occur because of a collision.

7.3 Game Data

The Torque game engine is designed to use many different types of data. Each type of data is recognized by its suffix.

- .cs – Torquescript files, converted to .dso object files on startup (text).
- .cfg – Used to define which nodes of an object (e.g. player character) to export to the engine (text).
- .dif – Compiled *interior* files.
- .dso – Compiled torquescript (.cs) and GUI (.gui) files.
- .dsq – Animation file for an NPC or a character.
- .dts – Shape files. Used for object, NPC, and player definitions.
- .dml – Sky definition file: defines the textures and cloud features for the sky box (text).
- .gui – Graphical user interface definition files (text).
- .jpg – Image format.
- .map – *Interior* definition file (text). Used to define non-convex shapes a user can enter (text).
- .max – 3D Studio Max file (not recognized by the game engine, but sometimes stored with it).
- .mis – Mission definition file (text). Sets up the game environment, initial objects, and many parameters (text).
- .png – Image format, usually used for textures (can include an alpha channel).
- .psd – Particle system definition file.
- .ter – Terrain definition files.
- .wav – Sound files.

Most of your game content will be in the form of .dts shape files, .map/.dif interior files, and .png texture files. The GUI (.gui) and terrain (.ter) files you will create once, most likely using the game engine interface. Your main characters will also require .dsq animation files in order to move appropriately.

It is possible you may want to understand the text file formats, such as the mission file, interior, and GUI formats. Depending upon the complexity of your game environment, you may want to populate the world algorithmically using a separate program that manipulates the content files directly.

Note: when I modified the GUI for the basic game setup, I lost mouse control over the character. It turns out there are some fields, with default values, that are not written to the GUI file when you save it unless the `canSaveDynamicFields` box is checked for the top-level GUI object. One of those dynamic fields is the `noCursor` field, which determines whether the mouse is visible or not. If the `noCursor` field has the value “1”, then the mouse events are capture by the engine and the behavior is correct. Otherwise, the user has control of the mouse and the engine does not capture the events.

7.4 Game Editors

Go through the tutorial that comes with the game engine: `gettingStarted.pdf`

7.4.1 Terrain Editor

The terrain editor is fairly self-explanatory. In-class demo.

Things to note:

- The default motion for the global camera is very fast, you may want to slow it down.
- Use Terrain painter to bring in terrain textures and brush them onto the ground.
- Using the Terrain editor, pick a brush, pick an action (add dirt/excavate/etc) and go.
- Swap between blue guy FPS view and a global camera you can move around.
- It's good to have flat ground where you want to put objects or players.

7.4.2 Mission Editor

- World Editor gives you a full screen view of your world and lets you manipulate objects.
- World Editor Inspector lets you view and manipulate object attributes. It also lets you organize objects into SimGroups.
- World Editor Creator lets you add new objects to the scene and create SimGroups (MissionObjects:System).
- Mission Area Editor lets you adjust the size of the mission area, center and mirror terrain

7.4.3 GUI Editor

The GUI editor is more complicated, but not that difficult to deal with.

Things to note:

- Make sure the "SaveDynamicFields" button is checked on the PlayerGui interface, otherwise it will not save the noCursor field and the mouse events will not be captured by the game engine.
- How you anchor your boxes and controls makes a big difference in how they move when the GUI is resized. Try out your GUI and multiple resolutions to make sure you're happy with it.

7.5 Torque Script

The Torque scripting language, torquescript, is the primary method of programming the engine and implementing a game.

7.5.1 Torquescript data

Torquescript is C/C++ with no types. Everything is an object, but there are different kinds of objects, and they behave differently with respect to how information is passed between the client and server.

You need to understand the client-server model because different kinds of objects move differently between the server and client.

Strings

- Tagged strings: single quotes, static
- Standard string: double quotes, sent every time

Example:

```
Thing1.property = 'Hobbit'; // This string could never change
Thing1.property = "Hobbit"; // This string could change
```

Objects: instance of an object class

- Objects have unique numeric identifiers– you can refer to these in script if you know them.
- Objects can have names– you can refer to these in script, but they may not be unique.
- You can refer to objects using local variables (%).
- Server and client have separate handles to each object, but the object information stays in synch.

Example: Each of the following do the same thing.

```
Thing1.property = 'Hobbit';

"Thing1".property = 'Hobbit';

%thingname = Thing1;
%thingname.property = 'Hobbit';

%thingname = "Thing1";
%thingname.property = 'Hobbit';
```

You can define new kinds of objects, but you almost always use inheritance: everything you want to make generally needs to inherit how to do basic things.

Datablocks: static information about a class

- Objects can have associated datablocks
- Datablock information is sent only once at mission startup
- Only object information not in the datablock is ever transmitted over the network
- Datablocks are a useful way to provide basic information about an object type.

Example:

```
datablock ClassIdentifier(NameIdentifier)
{
    AssignmentStatements;
}
```

The *ClassIdentifier* must be the name of an existing datablock class. There are a number of basic classes, including *PlayerData* and *VehicleData*, which contain a predefined set of fields used by the game engine to parameterize those kinds of objects. The *NameIdentifier* is the name you define for the datablock, generally something like *MyPlayersDatablock*. The assignment statements can set fields in the inherited class or you can create new fields for your particular object (which is the whole point of creating a new datablock in the first place).

Objects that are static and never change don't have datablocks (would be redundant).

7.5.2 Torquescript Functionality

Console: use it

Communicating with the game engine

global variables

object types

inserting objects via programming

Typical callbacks: examples

messages to objects

message to server/client

8 Game Interfaces

Game interfaces form a small, but important part of human-computer interface design. Unfortunately, there has been little use of academic HCI design principles in the computer game industry and little academic analysis or appreciation for the design methods of the game industry. The result is that computer game design has been largely trial and error (with many, many errors) with ad hoc (but often extensive) usability evaluation built into the game design process. Good games and good interfaces are often a result of *genius design* where a creative individual generates a successful design through intuition and perseverance. Such a method is difficult to teach.

8.1 Human Computer Interface Design Principles

There are many books, papers, and documents on HCI design. The following are a few design principles from one of them.

Cultivate sensitivity to design: look at designed objects in the world around you. For computer game design, examine the design of the interface, but look at it from the context of how design decisions help the user to accomplish their task. Interface design is system design, not something built in isolation.

Interact with the task, not the computer. The user's goal is to accomplish a task in a virtual world that they see, hear, and interact with via the computer. The user is not moving a mouse, they are moving the point of view of the character, or manipulating an object.

The game is the world in which the user wants to live. The computer is one way of accessing that world. We can also access game worlds through board games, imagination, and physical motion. What the computer provides is automatic management of complex state variables. The graphics and audio provide additional stimuli that can provide a greater sense of immediacy and awareness of the game world, but the game world itself is a vision of the game designer.

Do user testing first. User testing when a product is close to completion has little impact on the final product and any valid concerns are viewed with suspicion as excessive criticism. This fits in with the iterative design process suggested by Salen and Zimmerman.

The designer is NOT a typical user. Designers know too much about the game to have a good sense of usability. Having users unfamiliar with the game test the interface and game design is critical, as is listening to what they say.

Listen to user comments. Don't make your response to someone's comment about your design start with, "Well, I decided to do it that way because..." The user doesn't care why you made a certain design decision, and by explaining it to them you are not listening to their point of view. Whatever the reason, you may have made an incorrect decision and the time to fix it is now, not later. You will only be able to evaluate your design decisions by listening.

Technically, when designing a game it is important that the interface be modular and separate from the game state and game play. Cool looking scrollbars, or a novel USB keypad for your game are potentially interesting aspects of the interface, but they should have no impact on the game design. What they can impact is the immersive quality of the game. *Guitar Hero* is not the same game using a keypad, but the interface device has no impact on the actual mechanics or code of the game world.

8.2 Game Interfaces and Genre

Cultural and genre expectations impact computer game interfaces. In some cases, game interface elements may not be the best possible, but the culture of the genre and player expectations make it difficult to change them. Nevertheless, innovation in game interfaces is still possible (e.g. *Guitar Hero*).

8.2.1 First-Person Shooter

Examples: *Doom*, *Quake*, *Unreal Tournament*

Major interface elements:

- First-person video stream
- Visual proprioceptive indicators (see hands and weapon)
- Internal status indicators: health, ammo
- Targeting cross-hairs
- Heads-up display elements: map, enemy locations
- wasd keyboard input plus a mouse to provide directionality
- Proprioceptive indicators via overall video modification: reddish tinge, shaky camera
- Audio indicators of external action and change of internal status
- Chat area

There are few, if any, menus, dialogs, or other interface elements outside of the main video screen. Most external inputs are provided via the video stream. Most internal inputs are provided via status meters on the edge of the video stream, but some are indicated as modifiers to or active elements of the main video stream.

8.2.2 Adventure, Action

Examples: *Diablo*

Major interface elements:

- 3rd-person video stream, sometimes modifiable
- Visual proprioceptive indicators on the main character: armor, weapon, health
- Internal status indicators: health, mana, inventory
- Heads-up display elements: map, enemy locations
- Mouse input provides major motion and default action commands
- Multiple keys, some mapped by user, provide specific action options
- Rare modification of overall video stream (shaking, sometimes)
- Additional menus and dialog boxes to access inventory, internal status, companion status

- Audio indicators of actions and action results (player character getting hit)

The extra dialogs and menus are possible, in part, because the action is episodic in nature and there are periodic lulls where a player can safely ignore the main video stream and external events for short periods of time.

8.2.3 Strategy, Turn-based

Examples: *Civilization*, *Spaceward Ho!*

Major interface elements:

- Main video stream is an isometric view of a geographical area
- Overall status of strategic elements indicated by iconic representations
- Additional dialogs associated with most strategic elements
- Dialogs for handling overall game strategy elements (e.g. form of government)
- Mouse and key combinations to move strategic elements
- User-selected and arranged windows for indicating status of key elements
- Flexibility as to what information is provided in each window
- Little, if any audio associated with interface
- Interface for communication between players (not part of the main interface)

The lack of immediate time pressure makes it possible to give the user access to multiple dialogs and a large number of options and menu items. The challenge is to make common actions fast while also providing easy modification to enable specific user strategies. Interfaces tend to vary more in this class than in others because of the specifics of each particular strategy situation.

It seems like this genre is still in need of innovation.

8.2.4 Strategy, Real-time

Examples: *Starcraft*, *Dune*, *Warcraft*

Major interface elements:

- Isometric or 3D video stream showing characters and immediate surroundings
- Inset overview map showing known/unknown and allied/enemy units
- Mouse-based selection, motion, and simple action commands
- Key-based action commands with user mapping for specific key commands (e.g. hot locations)
- Object-specific menus accessed via mouse clicks on the objects
- Global status indicators: resources, gold, number of units
- Menus for overall strategy options

- Dialogs for overall action choices (building)
- Audio indicators of actions
- Chat area for communication with allies

The real-time nature of the game makes dialog chains short and the hot-keys an important part of the interface. Management of a large number of units makes mapping keys to unit groups critical, and players who learn the action keys have an advantage over players who have to use the mouse to select actions. Audio plays an important role in the interface, cuing players to events that happen outside the visible area.

RTS offers many challenges to the designer, as the player must effectively manage a large number of units in real time without having immediate visual feedback on every unit.

8.2.5 Adventure, RPG

Examples: *Realmz* ?

Major interface elements:

- Isometric or 3D video stream showing characters and immediate surroundings
- Dialogs, often numerous, for player status and action selection (exp. turn-based RPGs)
- Primarily mouse-based interface, with some keyboard shortcuts

How different from Adventure, Action?

8.2.6 Physical Skill

Examples: *Dance*, *Dance Revolution*, *Guitar Hero*, *Frogger*, *Olympic Decathlon*

Major interface elements:

- Video stream providing indicator as to next move
- Method of recording physical user action
- Overall game selection menus, but few interface elements during play

Board games exist for the same genre (e.g. *Blink*). The point of the game is to challenge the physical skills and hand-eye/foot-eye coordination.

8.2.7 Sports and Racing

Examples: *Grand Turismo*, *Madden NFL*, *NBA Basketball*

The sports and racing genre shares a lot with the FPS and physical skill genres. In addition, a game like *Madden NFL* also offers a turn-based strategy type feel with a large number of menus and options the user must select from for each play. As a genre, there is tremendous variety.

8.3 Game Design and Remote Robot Operation

Teleoperation of a robot is almost identical to a video game, with the exception of the robot existing in the real world, subject to real physics, and real uncertainty. There is a small, but growing literature on the design of remote operation robot interfaces that is of interest to video game designers. Note that a video game can always provide more, and more accurate information than a real robot interface because the video game world has a completely known state.

One of the primary concerns in remote robot operation is situational awareness. Situational awareness is the operator's sense of the robot's surroundings. The concept is essential to remote robot operation because an operator can make mistakes—such as colliding with the environment—if they do not have good situational awareness. A game player must also have good situational awareness in order to be successful.

Situational awareness has two parts: awareness of the robot's external surroundings and awareness of the robot's internal state. Given that many games are similar to the situation of remote robot operation, these concepts apply equally well to games and remote operation.

In remote robot operation, environmental awareness is often provided solely by video from a single camera. Simple visibility constraints often limit the situational awareness of the user.

Games often provide additional environmental cues in the form of overlay maps, inset maps with enemies and friends indicated as icons, crosshairs to indicate the destination of an action, or flashing objects indicating interesting attributes.

8.3.1 Analysis of Robotics Interfaces

A small number of studies of robot remote operation interfaces exist.

Geo-referenced camera (tied to being able to see the robot's chassis).

3rd-person v. 1st-person cameras (3rd-person is better)

Front and rear cameras (swapping driving direction automatically)

Sliding autonomy: idea that user can switch into different modes, providing different types of control. What about an FPS where you control multiple people with different modes of autonomy? New genre? Requires some AI.

Physical interface: joystick is a much more natural interface element than wasd for remote robot operation.

8.3.2 User Studies

1. Figure out what interface options you want to evaluate (keep it small, like 2-3).
2. Design a task that provides a challenge to the user: results must be discriminative.
3. Calculate the number of users required to eliminate ordering/learning bias.
4. Develop assessment criteria and mechanisms (e.g. forms)
5. Identify how all critical inputs will be recorded, ideally in a form permitting easy searching and access.
6. Develop assessment protocol.
7. Recruit users (at least 2-3 more than the minimum required)
8. Run first 2-3 users and evaluate the protocol, data capture, and discriminability of the results.
9. Revise protocol, if necessary (it probably is).
10. Run remaining users.
11. Evaluate data using accepted statistical evaluation techniques (look at literature for the field).

Assessment mechanisms are one of the most important parts of the process. There are numerous methods one can use.

- Quantitative user evaluation forms (check 1-4 for various questions).
- Qualitative user evaluation forms (describe the most useful part of the interface).
- Have user provide a stream of consciousness during the activity (record and transcribe speech).
- Record and quantify user actions, converting video into a symbolic sequence.
- Manually or automatically record specific events (e.g. collisions).
- Record keystrokes, mouse movement, other user actions with respect to certain tasks.
- Record user attributes (e.g. pulse, skin conductivity).

The key to assessment is to pick the tool that most directly measures the quantity of interest. If you want a game that generates excitement in the user, measure pulse and skin conductivity. If want to evaluate particular interface elements, develop a quantitative measure of performance and see which interface design results in superior performance.

8.4 Torque GUI

Overall GUI principles

Widgets: main widgets of interest

How much help do you need?

Name the widgets so you can refer to them in torquescript.

9 Modeling and Simulation

Modeling and simulation are two important parts of many programs. Even an FPS might want to model weather in a way that feels right to the player.

9.1 Agent-based Simulation

Sources:

- J. Epstein and R. Axtell, *Growing Artificial Societies*, MIT Press, 1992.
- E. Bonabeau, “Agent-based modeling: Methods and techniques for simulating human systems”, *Proc. of the National Academy of Sciences*, 99(3), 2002.
- R. Axelrod and L. Tesfatsion, “A guide for newcomers to agent-based modeling in the social sciences”, to appear in *Handbook of Computation Economics, v2: Agent-based Computational Economics*, Tesfatsion and Judd, ed.

Agent-based simulation is based on the idea of modeling systems using models of their constituent parts. Micro-economics, for example, is based on a simple model of the consumer. The alternative to agent-based simulation is macro-simulation where models represent global system behavior. There are, of course, a whole range of simulation strategies that incorporate both approaches, dividing the system as appropriate for the task and modeling it at a variety of levels of abstraction.

The benefits of agent-based modeling [ABM] are threefold (Bonabeau, 2002).

1. ABM demonstrates and explains emergent phenomena.
2. ABM provides a description of a system based on its constituent parts.
3. ABM is broadly applicable and simple to manipulate.

While the first item it is probably the most important feature of ABM, the other features help make ABM an important modeling tool. In particular, when trying to model a complex system, describing the system in terms of simple constituent parts is often the only feasible method. In many situations, it is also the only approach where the system parameters are easily describable and can be set using measurements over populations.

Research in ABM has four specific scientific goals (among others) (Axelrod and Tesfatsion, 2007).

1. Empirical understanding of systems: why do certain regularities or large-scale features of systems persist despite the lack of top-down management of the systems (or in spite of it)?
2. Normative understanding of systems: how can we use ABM to design system features that improve the performance of the system as a whole?
3. Heuristic understanding: systems, especially networks of individuals, can display complex, emergent phenomena not predicted by the constituent components. How can ABM help us to understand this behavior?
4. Methodological advancement: ABM provides a new methodology for analyzing systems, yet there is a lot about the methodology itself, and how to apply it, that we still have yet to discover.

Exercise 1:

1. Behavior 1: Everyone pick two people in the room, A and B, but don't tell anyone who they are. Move so as to keep A between you and B.
2. Behavior 2: For the same people A and B, move so as to keep yourself in between A and B.

Exercise 2:

1. Pick one of the following attributes: color of shirt, color of pants, or hair length.
2. Move so as to be near two people with similar qualities for the attribute you chose.

ABM is most appropriate when the behavior of individual system elements is nonlinear or not well-modeled by differential equations. Examples include:

- if-then rules, thresholds, or Boolean combinations of factors,
- agent behavior is stochastic in some form,
- agent behavior is heterogeneous at the level of the individual,
- agent's have the ability to learn or evolve their behavior over time,
- path dependence or storage of state information at the individual system element level,
- agent interactions are heterogeneous (can take many forms)
- agent interactions are topology dependent, and
- average interactions are inappropriate models of global system behavior.

It is not difficult to generate systems that exhibit the above characteristics, especially systems such as human societies.

ABM has been applied to many situations, and is currently finding a lot of application in business. For example, it has been used to evaluate (Bonabeau, 2002):

- flows: building evacuations, traffic jams, and customer flow under constraints,
- markets: stock markets, strategic simulation, and software agents,
- organizations: operational risk, organization design, and flow of communication, and
- diffusion: diffusion of innovation and adoption dynamics.

Classic examples of adoption studies include the QWERTY keyboard and Betamax v. VHS. For the latter, a simple model of adoption at the agent-level demonstrates how a de facto standard can emerge from small factors that may actually be non-optimal in terms of technological performance.

Example: flow management for evacuation.

Consider the situation where a large number of people are trying to exit a room through a single door. A simple model of behavior is for each individual to try and reach the door as quickly as possible. Also required is a model of when someone might be injured during the process. ABM is one method evaluating various flow management strategies.

For example, consider putting a column near the door (e.g. 1m), slightly asymmetric relative to the center of the door. Both ABM models and actual experiments with people in a room have

demonstrated that a greater number of people manage to exit the room without injury when the column is there compared to when there are no obstacles near the door. It is an unintuitive result, but one that can be repeatedly modeled using ABM.

The major drawback of ABM is the difficulty of evaluating the output of ABM simulations. While qualitative evaluations can be extremely revealing, extracting useful quantitative information is difficult. Typically, extracting any quantitative information from an emergent system first requires defining end-conditions for the simulation as well as a quantitative measure of structure. The actual quantitative information comes from running the system many times, varying input conditions and agent parameters, and correlating the outputs over many runs.

9.1.1 ABM Applied to Games

The benefit of using ABM in a game situation is that we can take advantage of all the good things—easy system development, intuitive parameter settings at the agent level—without having to worry about obtaining any quantitative information from the simulation. Our interest is only to simulate actual phenomena at the qualitative level: we want it to look right and act right and nothing else matters.

Areas of application in games include:

- environment/terrain development: simulating ecosystems, habitats, or landscapes,
- flora and fauna: simulating groups of critters and plants in the environment,
- NPC behavior: simulating crowds (small or large), armies, or other groups of NPC elements,
- markets: simulating the effectiveness of player decisions by modeling consumer choices,
- economies: simulating economic conditions by relating user-controlled variables to the behavior of simulated individuals,
- political systems: simulating the political behavior of the system by modeling individual behavior using simple models of voting behavior built on top of a topology representing communication channels.

ABM examples: like/unlike example of distribution of nodes to simulate both segregation and town-building. For example, if you have a town with a population that is divided by some characteristic (elves and dwarves or short and tall, or rich and poor) you can automatically lay out the town by running a simulation of individuals choosing where to live based on the rule that a majority of the nearby individuals should be similar in character.

For an economic model, you can imagine simulating an economy by have a thousand individuals with a typical wealth distribution (power law), spending habits that are dependent on circumstances, and a distribution of jobs (income streams for individuals). A set of simple rules would determine the spending and income for each individual at each time frame, and aggregate spending would be a statistic used to represent economic activity during the time period. Drivers to the economy could be the creation and loss of jobs, price fluctuations on goods, and the rate of change of the wealth of individuals. The difficulty would be balancing the factors so that swings in economic activity stay within reasonable boundaries.

10 Game AI

Maxwell's rule for AI entities

Do about the right thing most of the time. Avoid doing the wrong thing all of the time.

If we want to have realistic NPCs or AIs in our game, they need to react appropriately to the world. There are three pieces required for generating appropriate behavior: perception, planning, and action.

Whether the AI is running an empire or simply a demon waiting to jump out as you pass by, the entity must have an appropriate perception of the state of the world. The strategy AI should have access to the same type of information the player has and should not be able to reason based on elements of the game state that a human player cannot see. Likewise, the demon AI should not have perfect knowledge of the player's location and state, but should react appropriately to visual or aural cues such as seeing or hearing the player approach. Of course, the AI can cheat and access illegal information to give it an advantage against a human player, but human players hate that.

Note that the above comments assume a conceptual separation between the AI component of the game code and the game engine or code that converts the game state into perceptions. In some cases there is almost no separation between the two, and the limitations on perception are built directly into the AI component. For example, the code for the demon may be continuously accessing the player's position but only react when the player comes within a certain distance.

Planning is the component of the AI that determines the action or sequence of actions it will take. The planning can be extremely simple, such as a control law or simple reactive behavior in the case of the demon, or as complex as time allows, such as the AI for a strategy or chess game. There are a number of methods for planning AI actions, with different levels of complexity being appropriate for different types of games.

Action is generating the actual commands for the AI entity. What the entity does may be a simple function of the planning stage, or it may be a complex combination of factors to generate a single command. The action stage may need to be an arbiter of several different suggestions for action.

There are many ways to implement the three pieces of the control loop—perception, planning, and action. Implementing them as a serial pipeline is straightforward, but for most games the real-time nature of the system limits the complexity of the planning stage. Robot software architectures provide an alternative implementation using multiple layers.

For individual agents (NPCs that run around and interact with the world) robotics has some inspiration:

Behavior-based: sense-act paradigm, some kind of arbitration to determine action

Multi-layer architectures: sense-act low level, but more sophisticated planning higher up

For strategy-type agents, traditional AI techniques and learning provide guidance

- search type strategy techniques (state space & search)
- game-state evaluation strategy techniques (learning)
- hybrid techniques (learn states AND search several moves ahead)

10.1 Perception

Perfect state knowledge encoded by the game engine. Two approaches to making the AI appear like a human player are the following.

- Partial state access: the AI code is intentionally shielded from the complete game state, or is fed information that incorporates noise into the values. Partial state access is fast, but does not necessarily emulate the way a human is able to access game state information.
- Derived perception: the AI code must derive sensor information in manner that simulates the way humans derive sensor information. For example, the AI can only see the player when nothing is blocking visibility. Derived perception more directly emulates a human's access to game information, but can be very expensive to implement.

Note that, whichever method is used, it's all part of the same code. The barriers are artificial at best. However, it is useful to think of the AI scripting as having access only to derived perception events, effectively hiding the overall game state from the AI.

10.2 Action Planning

Behavioral/Reactive Systems: A simple approach to AI action selection is to derive a set of simple rules that determine the action of the AI as a function of its environment. For example, in the classic dungeon exploration game, monsters are often confined to a particular room. If the player enters the room, the monsters move towards the player and attack. If the player is not in the room, the monster goes idle, either standing still or move along a preset path.

```
if (player is close enough)
    execute attack action
else
    execute idle action
```

There are two straightforward ways to implement such a reactive system. The first is to use a derived visibility perception filter to trigger the behavior. If the player is “visible”, then attack. The other is to use a trigger space in the game that activates when the player enters it. In the case of the dungeon, the trigger could be the room itself.

In both cases, we want to avoid running the above code every tick of the game state, especially if we have hundreds or thousands of monsters in the dungeon, all evaluating whether to stand still or attack. The goal is to have only those monsters close to the player or immediately involved in conflict evaluating their activity functions. The event-based structure of the Torque engine makes that possible.

The conceptually complex feature of subsumption architectures that is important to consider when designing reactive architectures is that the world itself is the state vector for the entity. The AI itself does not store any state information or use internal state information to make decisions. Within a game world, the game engine stores the state of the world, and the AI code simply accesses the game engine data dynamically to make decisions. As noted above, it's how the AI is able to access the world state that makes it more or less realistic. When the world is the AI's state vector, then the AI's sensors become its method of accessing the state vector.

The idea of the subsumption architecture is that there are many simple processes running in parallel (division of computation) and that some combination of those processes gets control the AI's degrees of freedom.

In the base subsumption architecture, only the process with the highest current priority gets control. For example, for navigation through a set of obstacles, the AI could use the following set of rules.

- Forward motion in the absence of obstacles
- Obstacle avoidance through local potential fields when obstacles are in range of the sensors
- Wall-following when close to an obstacle (bug-style algorithm)

Different variations of behavioral architectures combine these in different ways.

- Subsumption: one behavior takes over based on current sensor inputs
- Fuzzy logic: behavior outputs get combined using a fuzzy control system
- Case-based: sensor states are divided into cases (crisp system)
- ANN: low-level behaviors may be trained using ANNs
- ANN outputs may be combined using any of above methods

Design task: design an AI for an NPC in an FPS game with short and long-range low-speed weapons.

10.2.1 Triggers

Triggers are bounding boxes in the game world that generate a callback when the player enters or leaves the box. Triggers also have a mechanism for setting up a callback function that executes at regular intervals so long as the player is still in the box. For the dungeon example, consider a trigger that constitutes a single room containing a monster. In the Mission Editor object tree, all monsters in the room should be part of the same group as the trigger. Figure 2 shows skeleton Torquescript code for this case.

The alternative to using triggers is to have a recurring method associated with the player character that periodically evaluates which monsters are in the area and starts a recurring method on the monster when the player comes close enough.

In the Torque engine, the simplest method of moving towards the player in this situation is to use the AIPlayer class function `moveToNode`, and give it the player's position as the goal. In the absence of that capability, a simple approach is to set up a simple control scheme that rotates the monster towards the player and moves it forward at a velocity proportional to the distance between the player and the monster. It is also possible to modify the `AIPlayer::moveToNode` function in the AI game engine to implement different low-level control strategies.

```
datablock TriggerData( RoomTrigger )
    tickPeriodMS = 500;
};

function RoomTrigger::onEnterTrigger( %this, %trigger, %obj ) {

    // the default C++ onEnterTrigger function calls the
    // function on all objects in the same group

    Parent::onEnterTrigger( %this, %trigger, %obj );
}

function RoomTrigger::onLeaveTrigger( %this, %trigger, %obj ) {

    Parent::onLeaveTrigger( %this, %trigger, %obj );
}

function RoomTrigger::onTickTrigger( %this, %trigger ) {

    Parent::onTickTrigger(%this, %trigger);
}

function MyMonster::onEnterTrigger( %this, %trigger, %obj ) {
    // %obj is the player data

    // probably want to store a reference to %obj in the monster data structure
    %this.playerToAttack = %obj;

    // tell the monster to move towards the player
}

function MyMonster::onLeaveTrigger( %this, %trigger, %obj) {
    // %obj is the player data

    // if the monster should stop attacking, have it enter the idle state
    // or go back to a path following task
}

function MyMonster::onTickTrigger(%this, %trigger) {

    // if the monster is close enough to the player, attack
    // otherwise move towards the player
    // the player to attack is stored in the monster data structure
}
```

Figure 2: Code showing how to implement a simple reactive system using a trigger area.

10.3 Achieving a Pose

A more natural method of achieving a pose in a space with no obstacles is to use a control law appropriate for a differential drive robot. We can treat any player as a differential drive robot with respect to motion planning, dividing the motion commands into an angular velocity ω and a translational velocity v .

The following is based on the derivation from Siegwart and Nourbakhsh, 2004, section 3.6.2.

The player's local frame of reference is $[x_r \ y_r \ \theta_r]^t = [0 \ 0 \ 0]^t$. If the player's goal is to reach an arbitrary location $[x_g \ y_g \ \theta_g]^t$, then we want to design a control law such that the difference, or error, between the player's current position and goal position goes to zero.

$$E = \begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix} - \begin{bmatrix} x_g \\ y_g \\ \theta_g \end{bmatrix} \quad (1)$$

Using a proportional control law, we want to design a system where the control velocities, $v(t)$ and $\omega(t)$ are a function of the error E .

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} k_{vx} & k_{vy} & k_{v\theta} \\ k_{\omega x} & k_{\omega y} & k_{\omega\theta} \end{bmatrix} E \quad (2)$$

Mathematically, the idea of moving to a specific goal location relative to the player's frame of reference is identical to defining the goal as the origin of a global coordinate frame. The kinematic equation for differential drive motion in the global coordinate frame is given by (3)

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 \\ \sin \theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \quad (3)$$

If we transform the description of the player's position into polar coordinates, it turns out to be simpler to define the control law. The global kinematic equation in polar coordinates is given in (4), where

- ρ is the distance from the goal to the player's center,
- β is the angle of the line connecting the player to the goal point relative to the x -axis, and
- α is the orientation of the player relative to the line connecting the player to the goal point.

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -\cos \alpha & 0 \\ (\frac{1}{\rho}) \sin \alpha & -1 \\ -(\frac{1}{\rho}) \sin \alpha & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \quad (4)$$

Now consider the control law in (2), but in polar coordinates and with a single constant for each aspect of the player's pose in polar coordinates.

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} k_{\rho} \rho \\ k_{\alpha} \alpha + k_{\beta} \beta \end{bmatrix} = \begin{bmatrix} k_{\rho} & 0 & 0 \\ 0 & k_{\alpha} & k_{\beta} \end{bmatrix} \begin{bmatrix} \rho \\ \alpha \\ \beta \end{bmatrix} \quad (5)$$

If we substitute (5) into (4), then we get a control law that specifies the polar representation velocities required to drive the error to zero as a function of the player 's global position.

$$\begin{aligned} \begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} &= \begin{bmatrix} -\cos \alpha & 0 \\ (\frac{1}{\rho}) \sin \alpha & -1 \\ -(\frac{1}{\rho}) \sin \alpha & 0 \end{bmatrix} \begin{bmatrix} k_{\rho} \rho \\ k_{\alpha} \alpha + k_{\beta} \beta \end{bmatrix} \\ &= \begin{bmatrix} -k_{\rho} \rho \cos \alpha \\ k_{\rho} \sin \alpha - k_{\alpha} \alpha - k_{\beta} \beta \\ -k_{\rho} \sin \alpha \end{bmatrix} \end{aligned} \quad (6)$$

Unlike the pure polar kinematic equation, which has a singularity at $\rho = 0$, the control law has no singularity and reaches stability ($\rho = \alpha = \beta = 0$) at the origin aligned with the x -axis.

Conditions for stability:

- The translational velocity $v(t)$ must always have the same sign throughout the trajectory.
- The angles α and β must always be expressed in the range $(-\pi, \pi)$.
- If α begins in the range $\alpha \in (-\frac{\pi}{2}, \frac{\pi}{2}]$ then the velocity term is positive and the player moves forward towards the goal.
- if α begins in the range $\alpha \in (-\pi, -\frac{\pi}{2}] \cup (\frac{\pi}{2}, \pi]$ then the velocity term is negative and the robot moves backwards towards the goal.
- $k_{\rho} > 0$ and $k_{\beta} < 0$ and $k_{\alpha} - k_{\rho} > 0$

Some values to try initially as control constants are $(k_{\rho}, k_{\alpha}, k_{\beta} = (3, 8, -1.5))$. One thing to be aware of, is that if the monster is facing away from the player in its initial configuration, then blind application of the equations will move it backwards towards the player. If such action is inappropriate, then rotate the monster in place until it is oriented more towards the player and then apply the control law.

10.4 State machines

Other, more sophisticated, reactive systems are possible, but all have the same form: a set of rules and an arbiter that determines which rule is currently active. The arbiter may be simple priority, often in the form of an *if-then-else-if-else* type structure. The arbiter may also be more sophisticated, or even learned, such as an artificial neural network. For simple "go-bash-it" style monsters, nothing more is required. However, AI characters that may need to execute a sequence of events, or that have more than one response to a situation, state information and state machines are useful tools.

Conceptually, a state machine is a sequential logic circuit with at least one input, memory (state), and at least one output. The output is a function of the current state and possibly the input. The next state of the circuit is a function of the current state and the input. The inputs are normally binary variables, but in some cases may have multiple discrete values. Input variables with a large number of possible values are difficult to handle in a simple state machine and require an additional layer of abstraction.

Example: Guard Demon

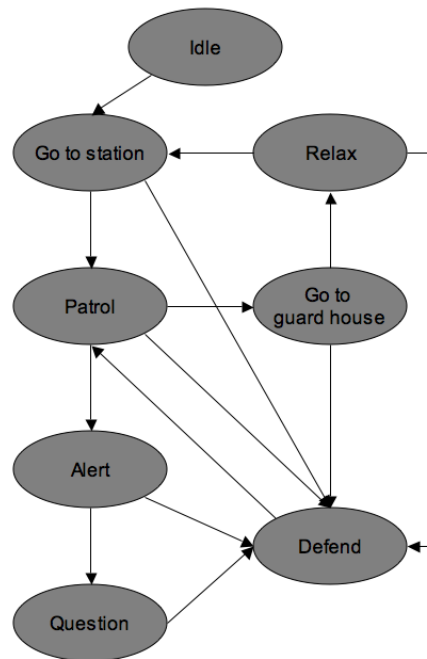


Figure 3: Guard state machine.

10.5 Emotions in AI Characters

Emotional state spaces as action modifiers

- Emotions can improve the situational behavior of the AI Player
- Emotional state can be mapped to a set of axes, typically 1, 2, or 3.

Mahrbian's PAD space: Pleasure, Arousal, Dominance

Events modify emotions

Emotions modify action selection and action parameters. For example:

- Spoken phrases
- Action decisions
- Quality of the actions
- Animations of basic motions

Robot Improv: fairly sophisticated modeling of how emotional state (inner obstacles) could affect behavior and action selection.

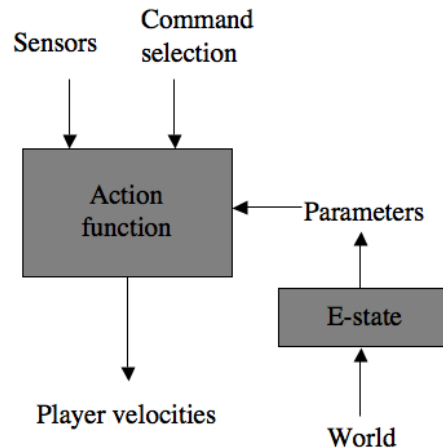


Figure 4: Model of the interaction between emotion and actions.

10.6 Motion planning

10.6.1 Visibility Graphs

Visibility graphs and retractions are like roadmaps of the world that tell the robot where it can and cannot go. By searching the maps, the system can identify one or more paths, if they exist, between the start and goal locations.

A standard visibility graph generally connects together the corners of obstacles. Visibility graphs are guaranteed to find shortest paths between locations in the world and correspond well to the way people navigate a situation.

Assumptions:

- A, B are polygonal
- $W = R^2$
- translational/holonomic robot, so $C = R^2$

The idea of the visibility graph is to construct a semi-free path as a polygonal line from the start location Q_0 to the goal location Q_F .

Proposition 1: Let CB be a polygonal region of $C = R^2$. There exists a semi-free path between any two given configurations Q_0 and Q_F if and only if there exists a simple polygonal line T lying in $cl(C_{free})$ whose endpoints are Q_0 and Q_F , and such that T 's vertices are vertices of CB .

Proof: If there exists a path, then there exists a shortest path. Locally, the shortest path must also be the shortest path. Therefore, the curvature of the path must be zero except at the vertices of CB .

Therefore, it is sufficient to consider the space of lines between obstacle vertices when planning a path. The nodes of the graph are Q_0 , Q_F , and the vertices of CB . The visibility graph itself consists of all line segments between the nodes that lie in C_{free} except for possibly their endpoints, in the case of Q_0 and Q_F .

In English, if you have a set of obstacles, you go around them as closely as possible at the corners. Since the configuration space has grown the obstacles, the corners are “safe” for the robot to traverse. The drawback is that you come close to obstacles and have to rely on the safety factor built into the obstacle growing step.

There are two methods one could use to design a visibility graph within a game. First, set up a series of markers in the world, identifying which ones are visible with respect the others (i.e. have no obstacles in between them). The graph of marker connectivity is, for all intents and purposes, a visibility graph.

The second method would be to obtain the bounding boxes of all obstacles within the relevant area. Then grow the obstacles by the radius of the player to guarantee that the player does not intersect with the bounding box corners when rounding an obstacle. Finally, use a standard visibility graph algorithm to generate the graph. Given that you only need to make the part of the graph that includes the static obstacles once, and off-line, the dumb-stupid $O(N^3)$ algorithm where you consider each possible edge between every pair of vertices would work just fine. To insert the dynamic player or a dynamic goal into the graph (both the player and the goal are nodes in the graph), just identify all obstacle corners that are visible from the player’s location ($O(N)$ task).

A **retraction** is a mapping from any location in a topological space X onto a subset of the space Y that preserves the topology of the space. A mapping $X \rightarrow Y$ is a retraction if and only if it is continuous and its restriction to Y is the identity map. (The piece of X that is in Y maps identically to Y).

Let P be a retraction. P preserves the connectivity of X if and only if for all $x \in X$, x and $P(x)$ belong to the same path-connected component.

Proposition 2: Let P be a connectivity-preserving retraction $C_{free} \rightarrow R$, where R , a subset of C_{free} , is a network of 1D-curves. There exists a free path between two free configurations Q_0 and Q_F if and only if there exists a path in R between $P(Q_0)$ and $P(Q_F)$.

What this says, is that if we have a retraction, we can plan motions on the retraction space and guarantee they are executable in the sense of the player not running into anything. Retractions are a bit tougher to generate, but there are computational geometry libraries available that could be used to build Voronoi diagrams from the set of bounding boxes. A Voronoi diagram is a set of paths that are equidistant from CB at all points. Therefore, it maximizes the distance between the robot and the obstacles. In a polygonal world, it will consist of lines and parabolic line segments. For motion planning for something like a horse and buggy, or a parade of soldiers, the Voronoi diagram would generate appropriate motion. For individuals, a visibility graph result will probably look better.

10.6.2 Dijkstra’s Algorithm

Dijkstra’s algorithm is a greedy algorithm for identifying the shortest path between two points in a graph. Nevertheless, it is still proven to provide the shortest path between a source node and any other node in the state space.

Dijkstra’s algorithm builds a **spanning tree**, which is a graph with no loops that gives the shortest path from each achievable node in the state space to the source node. Given a spanning tree, the shortest path to any node is possible to determine in time proportional to the length of the path.

Algorithm:

- $G = (V, E)$: a graph with a set of vertices V and edges E

- S: the set of vertices whose shortest paths from the source have already been determined
- Q: the remaining vertices
- d: the array of best estimates of shortest paths to each vertex (one element per vertex)
- pi: an array of predecessors for each vertex (one element per vertex)

```

shortest_paths( Graph g, Node s )
  initialize_single_source( g, s )
  S := { 0 }          /* Make S empty */
  Q := Vertices( g ) /* Put the vertices in a priority queue */
  while not Empty(Q)
    u := ExtractCheapest( Q );
    AddNode( S, u ); /* Add u to S */
    for each vertex v in Adjacent( u )
      relax( u, v, w )

/* u is the newest node added to the spanning tree */
relax(Graph g, Node u, Node v, double w[][] )
  if g.d[v] > g.d[u] + w[u,v] then
    g.d[v] := g.d[u] + w[u,v]
    g.pi[v] := u

/* Initialize the source node to a distance of 0, and rest to inf */
initialize_single_source( Graph g, Node s )
  for each vertex v in Vertices( g )
    g.d[v] := infinity
    g.pi[v] := nil
  g.d[s] := 0;

```

10.6.3 A* Search Algorithm

The A* search algorithm is a search method that takes into account both the estimated cost to the goal, h and the cost of the path so far, g . By using their sum, $g + h$ to decide which node to expand next, A* gains some nice properties. In particular, if h provides an exact or underestimate of the cost to the goal, then A* is guaranteed to find a path, if it exists, and the first path it finds will be the shortest path. Because A* takes into account the estimated cost of getting to the goal, it is more efficient than breadth-first, as it tends not to search nodes that do not bring it closer to the goal.

A search algorithm requires a **state space** in which the search takes place. A visibility graph, for example, represents a state space, where each node in the graph is one state. The **root node** is the starting node for the search, generally the robot's current location. The **children** of a node are all of the states accessible from the current node. In the case of the visibility graph, the child nodes are all of the nodes connected to the current node via an unblocked line.

In addition, A* requires the following components.

- OPEN list: sorted list of places to explore that are connected to places youve been
- CLOSED list: places you have already explored
- $g()$ function: cost from root to the current node

- $h()$ function: estimated cost from the current node to the goal

Algorithm

1. Put the root node on the OPEN list
2. While the OPEN list is not empty, extract the first item
 - Test if the node is the goal
 - If it is, terminate the search
 - return the path to the goal by linking through all the ancestors of the goal node
 - If it is not the goal, expand the node to find its children
 - For each child
 - If the child is on the CLOSED list
 - * Carefully consider what needs to happen
 - * If the node on the CLOSED list has a shorter g value, then delete the child
 - * Else, make the current node the parent of the node on the CLOSED list and replace the g value of it with the g value of this child and propagate the g value to all of the descendents of this node
 - If the child is on the OPEN list
 - * Carefully consider what to do if it is
 - * If the node on the OPEN list has a shorter g value, then delete the child
 - * Else, delete the node on the OPEN list and insert the child in the proper order
 - Else, if the child is not on either the OPEN or CLOSED lists
 - * Put the child on the OPEN list with a link to its parent
 - Place the current node on the CLOSED list
3. If the OPEN list is empty, return failure in the search

The characteristics of the functions g and h determine the behavior of the A^* algorithm.

- Breadth-first search: Estimate of cost to the goal (h) is zero
- Greedy/depth-first search: Estimate of cost from the root node (g) is zero
- A^* is optimal if the h function is exactly the cost to the goal
- A^* returns an optimal path if the h function under-estimates the distance to the goal
- A^* is pretty good if the h function only sometimes over-estimates the distance to the goal

Sometimes in robotics we are computation-limited with real time deadlines. In that case, we may not be able to find a complete solution from start to finish. However, it is still possible to make an estimate of which direction the robot ought to move to achieve the goal even given the hard time deadline. The method is called IDA*, or **iterative deepening A^*** search. The basic idea is to initialize a cost f that bounds the extent of the search and then use the following algorithm.

1. Run A* search until the next node to be expanded has a cost $g + h > f$.
2. If the goal is not found, store the path with the least cost to the goal h , increase f , and repeat. Otherwise, return the path.

IDA* is an anytime search algorithm because the process can be interrupted and there will still be a best guess available from the prior search. Since the search algorithm always expands most of the nodes at the bottom level, the cost increase is not significant compared to the total cost. Note that IDA* with g (path cost) set to zero becomes iterative deepening depth-first search. IDD search can be implemented easily with minimal memory requirements, and in some situations is appropriate.

10.7 Search with Opposing Players

When searching for moves in a game with two or more opponents, the differing goals of the opponents make the search more difficult. Player A wants to maximize its score and minimize player B's score, while player B wants to do the opposite. The branching factor in games makes it rare for a search tree to reach from the players' current positions to the end of the game, so some evaluation function is required to estimate the quality of a particular game state.

10.7.1 Evaluation Functions

Evaluation functions are the real trick in generating computer players. As the computer will be using all of its resources to optimize its score on the evaluation function, that score must reflect the actual quality of the position. Actions that lead to a high score, whether or not they actually lead to success in the game, will be the actions taken by the computer player.

There are two general methods of creating evaluation functions: manual design and learning. Manually designed evaluation functions are often useful as first-cut approximations. Sometimes, with adequate tuning, they are sufficient to provide a challenging opponent.

Learned evaluation functions are generally based upon a very large number of runs of the game with computer opponents playing one another. So long as the computer opponents have some evaluation function to work with, even if it is pretty bad, the computers can make semi-intelligent decisions and play realistic games. The important thing is to play lots and lots of games, possibly with many different starting conditions. Each game generates a sequence of moves with a winning and a losing sequence. Because of chance, or different move selection by one of the players, the same game state may result in a win in one sequence and a loss in another sequence. Alternatively, the same game state may result in a win in all of the sequences in which it occurs. By playing a large number of games, the system is able to collect statistics on a large number of game states.

One problem is that the number of possible game states is so large, that for most challenging games there is no possibility of collecting statistics on all of the states. Therefore, the system must have a method for learning that lets it make estimates for new states based on experience with similar game states.

The three major issues, therefore, in creating a learned evaluation function are:

- generating an estimate of state quality from its occurrence in one or more game sequences,
- generalizing the learned function to states not encountered during the training period, and
- generating lots of realistic game sequences.

One solution to the first issue is **reinforcement learning**, which is a method of assigning a value to a state based on its eventual outcome. The following is from (Boyan, 1992). Consider a sequence of states that lead to the outcome O_A .

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \dots \rightarrow x_N \quad (7)$$

A naive, or supervised approach to generating the value of the states $[x_0 \dots x_N]$ is to assign them all a value of O_A . That would generate the training pairs:

$$\{(P(x_0), O_A), (P(x_1), O_A), (P(x_2), O_A), (P(x_3), O_A), \dots, (P(x_N), O_A)\} \quad (8)$$

If one of the states, x_2 , for example, appeared on a different state sequence with outcome O_B , then the training pair $(P(x_2), O_B)$ would also exist in the training set. The problem is that the training set then becomes contradictory, and it is up to the specifics of the learning method as to how it will satisfy the training set constraints.

An alternative, called temporal difference methods, instead use the current predicted value of the next state to generate the current training set. A temporal difference method using only the next timestep's prediction, also called reinforcement learning, would generate the training pairs:

$$\{(P(x_0), P(x_1)), (P(x_1), P(x_2)), (P(x_2), P(x_3)), (P(x_3), P(x_4)), \dots, (P(x_N), O_A)\} \quad (9)$$

As the system plays new games and executes learning iterations, the latest prediction function is used to generate new training pairs. Only the final state in the sequence maintains its fixed value (generally winning or losing the game).

The second issue, generalizing to novel states, is generally accomplished by using a flexible learning mechanism such as a neural network that can generalize to new inputs. For example, a simple feed forward multi-layer perceptron can learn complex functions and generalize its learning to similar, but not identical states.

The third issue has two possible solutions: 1) use a hand-designed system to generate game sequences, or 2) let the learning system play against itself as it gets better. It turns out the latter tends to produce better performing systems, in general, unless the hand-designed system is exceptionally good to begin with.

In (Boyan, 92), for example, when he let the training neural networks play against themselves, training the network for a few epochs each time a new set of game sequences were generated, they performed better than when they were trained by playing against a hand-designed player. His hypothesis was that, when playing against themselves, the networks did not learn any specific trick or have any special advantage over their opponent.

Boyan's later work (Boyan, 96) showed that care must be taken in using function approximators for estimating state values. In particular, care must be taken as to which states to use in training the function approximator, and that simple backtracking as above can produce undesirable results, especially in areas of the state space that are not smooth. The concept he proposes, called *rollout* is to use for training only states for which the function approximator can reasonably estimate the values for a sequence of states leading to a known goal. As the training process continues, new states can be added to the training set when they can get to a state in the training set (the rollout set) in a single action.

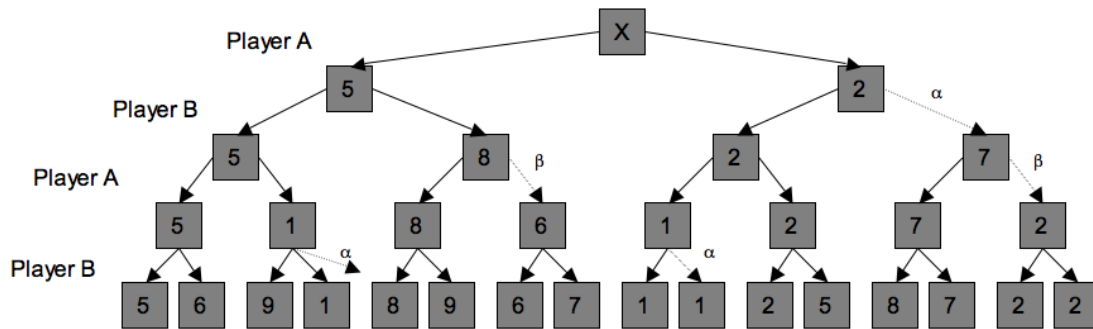


Figure 5: Minimax tree showing α and β prunings.

10.7.2 Alpha-Beta Search

Given a perfect evaluation function, search is unnecessary and a greedy play algorithm will work fine. However, evaluation functions are rarely perfect, and often, searching several steps forward can significantly improve the ability of the system as a whole to select appropriate actions.

Alpha-beta search is a method of reducing the amount of search required to find the best current course of action. The basic concept is that player A wants to maximize a score, while player B wants to minimize it. One of the basic assumptions of gameplay is that player B will always minimize the score given a set of choices. Therefore, unless a move by player A immediately results in a win, the value of a move by player A is only as good as the value of the move that minimizes the score on player B's next turn. The converse holds, however, player B can only minimize the score to the best subsequent option open to player A.

Figure 5 shows a possible play tree with two choices for each player at each turn.

Probabilistic games pose a more difficult challenge to reasoning about actions. Consider, for example, the game of Risk. Each state should represent a complete world state for the board with all of the armies, locations, and Risk cards for each player as well as information about the currently active player and stage of play. If Player A is trying to decide what to do, there are many possible options. Some options, like attacking, have probabilistic outcomes. There is a possibility, for example, that attacking 20 defenders with two attackers will result in a win, which could lead to a very good outcome for player A. However, most of the outcomes of that action would probably be bad. Somehow, we must incorporate the probabilities of outcomes into reasoning about likely next states in order to evaluate the quality of an action.

The alpha-beta pruning algorithm (from Russell-Norvig) is as follows.

alpha: best option available to player along path so far, init to $-\infty$
beta: best option available to opponent along path so far, init to $+\infty$

```
function Max-Value(state, game, alpha, beta) {
  if cutoffTest(state) then
    return Eval(state)

  for each child C of state
    alpha = max[ alpha, Min-Value(C, game, alpha, beta) ]
    if alpha >= beta then
      // opponent will never let the tree get to state
      return beta

  return alpha // best option to player
}

function Min-Value(state, game, alpha, beta) {
  if cutoffTest(state) then
    return Eval(state)

  for each child C of state
    beta = min[ beta, Max-Value(C, game, alpha, beta) ]
    if beta <= alpha then
      // player will never let the tree get to state
      return alpha

  return beta // best option to opponent
}
```

10.8 Online Learning

Learning is useful for a number of tasks in game design: low-level control of NPCs, game state evaluation for strategy and board games, or learning the parameter adjustment function for emotional states. One potentially interesting use of learning is to modify the game on-line in response to the user's actions or strategies. The question is how to use learning to generate a noticeable difference in the behavior of the game while the user is playing.

The issues in online learning are:

- Selecting appropriate game factors to be modified: what embodiment of the computer within the game is appropriate for learning, and what actions of the computer should be affected?
- Generating training data: what experiences within the game should be used as training data for the learning system? What events constitute positive and negative training examples?
- Setting appropriate learning parameters: can the system learn fast enough, while still being stable, to have an effect within the time period of the game?

Characteristics of a game that might be appropriate for learning include:

- Finite-state-machine transition probabilities: whether you have zombie or a guard, there are events that trigger actions or a change of state. In the case of a guard, if you shot at a guard the last time you

approached a tower, even if he guard you shot at is no more, the guards may have collectively learned to shoot first and ask questions later.

- User tendencies

11 Combat

Torque issue: Player bounding box (not set as part of the mesh)

Whacking things: can't play an animation and ask what did it hit

- If the opponent is close enough, use probabilities to assess hits and blocks
- User provides input as to when to assess hits and blocks for the player character
- Monster's need scheduling

Animating an action:

- Create the animation
- Name it something and put it in the same directory as the shape
- In player.cs, add the animation to the animation map. Given it a sequence number, the file path, and a name.
- In default.bind.cs, map a key to the action and create a function that calls a server command.
- In player.cs create the server function, which ought to call a Player class member function.
- In player.cs create the Player class member function that calls setActionThread

12 Lots of practical stuff

- Animations
- Sound
- ShapeBaseImageData and their state machines
- Projectiles
- Particle Systems
- Explosions
- Level of detail

13 Advanced Graphics Applications

13.1 Realistic Motion

C. Karen Liu and Z. Popović, "Synthesis of Complex Dynamic Character Motion from Simple Animations", SIGGRAPH 2002.

Synthesis of complex, realistic motion is difficult. While keyframes permit animators and artists to achieve a particular pose, simple interpolation of keyframes may not produce realistic motion. Inverse kinematics systems with dynamic constraints, while they produce physically correct motion, may not allow for sufficient artistic input.

Liu and Popović propose a solution that combines physical modeling with artistic keyframing in a way that makes realistic motion easier to achieve.

Their system has four stages.

- Constraint and stage detection: identify constraints provided by the environment by finding bone locations that do not move over a series of frames. Separate the animation sequence into constrained (figure is touching something that provides an external force) and unconstrained (figure is affected only by gravity).
- Transition pose generation: create a keyframe at each transition point between constrained and unconstrained motion. These frames can be manually supplied by the artist, or suggested by the system.
- Momentum control: generate constraints based on Newtonian physics, in particular conservation of momentum, and biomechanics observations about momentum shifts.
- Objective function generation: construct the overall bone motions so as to minimize an objective function, constrained by the external forces and momentum control, that favors smooth motion, similar to the input keyframes, and that is balanced when stationary.

Constraints and Stage Detection

The idea is to examine the bone motions to identify places where at least one point on the bone does not change location over a certain period of time. They first identify motion of points on a bone that are stationary (point contact), then look for points that fall on a line, and then look for points that fall on a plane. The problem reduces to a simple eigenvector computation.

Parameters to the system include:

- minimum frames required to identify a point as stationary,
- degree of tolerance for matching up point locations from frame to frame, and
- which body parts should be examined for constraints (e.g. feet and hands).

Based on the found constraints, the animation can be split up into constrained and unconstrained parts.

Transition Poses

The described system includes two possible methods of creating transition poses: 1) let the animator create them, and 2) create a database of typical poses and generate new poses from learned examples.

The system has a method of representing a pose by using three center of mass calculations: lower body, upper body, and arms. The locations of the COM positions are stored relative to the character's center of support,

making them invariant to global transformations. Note that the poses being learned are all transitional poses, and the character is necessarily touching at least one constrained surface.

Given a new transition pose, the system searches the database of stored transition poses, looking for similar mass distributions. It creates the new pose using a weighted blend of the top three matches. The animator can then modify the transition pose, if necessary.

Momentum Constraints

During unconstrained stages, the linear momentum of the object is affected only by gravity. The angular momentum is constant.

During constrained stages, changes in the linear and angular momentum are a result of complex interactions. The constrained stage can also be thought of as a transition between unconstrained stages, where the linear and angular momentum entering the constrained stage gets transformed to the momentum at the exit frame. Biomechanics studies have shown that momentum during a constrained stage tends to follow a specific trajectory where the momentum of the prior unconstrained motion is absorbed, and then the momentum of the new unconstrained stage is created, with a mild overshoot. The studies also show that the absorbing motion is larger than the overshoot on the generating motion.

By constraining only the linear and angular momentum to follow a biomechanically plausible path, the overall computation is much simpler.

Motion Generation

The final motion generation is the result of using a non-linear constraint solver to create the overall bone animations subject to the positional and momentum constraints as well as the keyframes provided by the animator. In addition to meeting the constraints, the factors used by the objective function are:

- Minimum mass displacement, which is akin to minimum energy usage.
- Minimal velocity of joint angles, to try and minimize abrupt transitions.
- Static balance, which is the distance of the center of mass from the constraint locations when projected onto the gravity plane.

13.2 Crowds

A. Musse, B. Ulicny, and A. Aubel, "Groups and Crowd Simulation", Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2004.

Crowd simulation has a number of difficult aspects:

- Character development and representation
- Character behaviors and decision-making
- Crowd generation and control
- Rendering

While these issues are generic to the creation of virtual characters, generating crowds of individuals magnifies the problems because of the large number of entities involved. Making any one aspect of crowd generation too complex bogs down the whole system.

Crowd simulation systems intended for guiding the design of spaces generally use cellular automata to represent individuals, providing rules for behavior and a distribution of individual attributes. Such systems must be validated against real data in order to be useful in the design process.

Crowd simulation systems intended for visual consumption need to look realistic, but also need to be controllable at a high level so that a visual creator can achieve the desired effect. The two goals of visual realism and validated results are not exclusive. Good visualizations help designers to understand crowd behavior, and validated crowd behavior looks realistic.

High end systems (e.g. LegionTM): Model each individual according to empirical distributions of parameters (e.g. walking speed), with decisions based upon local perception of the environment. Perceptions may even be derived from computer vision analysis of synthetic images. Such systems, however, need not be real time, as synthesized movies demonstrating the results of various design decisions are adequate. The visualizations are also generally not intended to be realistic, but only need to show flow patterns and answer questions like, "how many people got out?".

Systems intended for real-time rendering and manipulation need to minimize both design time and rendering time, which means that each individual in a crowd needs to be computationally lightweight.

13.3 Crowdbush

B. Ulicny, P. de Heras Ciechowski and D. Thalmann, "Crowdbush: Interactive Authoring of Real-time Crowd Scenes", Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2004.

Crowdbush is an example of a real-time rendering and crowd manipulation systems that balances a number of different factors.

Design: Characters are polygon meshes with skeletons. Typical figures have between 330-1100 triangles with multiple textures representing different body parts.

Behaviors: Reactive control, using a simple force-based system to keep individuals from colliding.

Creation: Use a brush abstraction to spray a crowd onto a scene. The brush acts like a 2D brush, but the characters get inserted into the 3D scene. In some cases, the characters are inserted into a pre-specified grid (such as might be necessary in a theatre), in other cases the locations are selected randomly according to a distribution defined by the brush. Individuals can be uniform or randomly drawn from a population.

Control: Use a brush to spray attributes or actions onto a crowd. Attributes such as emotions can control which animations get selected. Actions such as walk, or run, cause certain animations and motions to occur. The user can also specify a series of waypoints and have the characters follow the waypoints, with control over the width of the path within which the characters walk.

Rendering: Rendering in real time requires careful cache management. For each template model, the state of the model at each animation sequence gets stored as an OpenGL display list (e.g. a module). If a model has more than one texture, the model is divided into separate display lists by texture. All individuals using the same polygon templates get rendered together. Within that group, all characters with the same textures get rendered together.

13.4 Graphics Processing Units

13.4.1 Graphics Pipeline

The generic graphics pipeline has a series of operations that need to occur in order.

- Interface with the host CPU: downloading the required data and converting it into canonical formats.
- Geometry analysis: transformation from 3D world space into 3D clip space.
- Primitive Assembly/Setup: Clipping, mapping to 2D image space through division by the homogeneous coordinate, culling, and setup of pixel fragments.
- Color/Texture: Texture mapping, color, and depth generation for the pixel fragments.
- Framebuffer interface: Insertion of the pixel fragments into the framebuffer.

Up until about 2000, with the NVIDIA GeForce3 GPU, these stages were largely fixed on commercial graphics cards, and the programmer had control over only the inputs to the pipeline. Programmable GPUs existed, but were largely specialty items programmed using assembly code or at an even lower level for custom ASICs.

13.4.2 Programmable Vertex Shaders

With the advent of the GeForce3 GPU, graphics processing units provided graphics programmers with the ability to load small programs into specific stages of the graphics engine. In particular, they added a programmable stage in the geometry analysis stage called a **Vertex Shader** that could modify the characteristics of a vertex before passing it on to the remainder of the pipeline.

To enable vertex shaders, the system required:

- A virtual machine model, or Instruction Set Architecture [ISA]
- An instruction set
- The ability to download the programs

The initial vertex shaders used very simple virtual machines to maintain the flow of the parallel graphics pipeline and ensure no resource conflicts or serial dependencies could occur. In particular, no vertex calculation could depend upon another.

- All data consists of quad-float vectors (x, y, z, w) , using the IEEE 32-bit floating point format.
- The floating point core is approximately IEEE, but does not support de-normalized numbers or exceptions, and rounding is always towards negative infinity.
- All input data is read-only
- All output data is write-only
- The instruction set contains no branching (but it does have conditional set commands)
- There is a small register file
- All instructions take the same amount of time, and all inputs are always available.

- Each input can be **swizzled** or negated prior to a functional operation.

The GeForce3 vertex shader supported 17 operations, including some complex functions like distance, reciprocal square root, multiply and add, and Phong lighting. The ability to swizzle and negate operands eliminated the need for a specific cross-product or subtraction operation.

The vertex shaders allow for interesting effects, such as velvet (rotating normals 90 degrees), mesh deformations, bump-mapping, and environment-mapping within the GPU hardware.

13.4.3 New Programmable Pipelines

New programmable pipelines offer additional stages in the pipeline for user-defined programs. The Direct3D 10 engine, for example, has three stages that provide use programmability.

- Vertex shader: program receives as input a 3D vertex and outputs a 3D vertex
- Geometry shader: program receives as input a single primitive (point, line, triangle), and outputs zero or more primitives, up to a maximum amount of byte information.
- Fragment/Pixel shader: program receives as input a single pixel fragment and produces a single output fragment consisting of 1 to 8 attribute (color) values and optionally a depth value.

Unlike its predecessors, the Direct3D 10 engine provides a single virtual machine (ISA) for each programming stage. In addition, it provides a complete set of 32-bit integer instructions in addition to the traditional floating point instructions. The floating point instructions also more closely follow IEEE accuracy.