

Direct3D Project Report

3D Dominoes

May 2008

John Pile Jr
john@alaskajohn.com

Assignment:

You are to produce a 3D computer game incorporating the following features:

- a) Keyboard and mouse interaction*
- b) Audio*
- c) Simple physics*
- d) Use of an effects file*

In addition, your game (or its production) must incorporate knowledge or techniques gained from researching a "special technique".:

CS1122A Programming Games
Dr. Louis Natanson

MSc, Computer Games Technology
University of Abertay Dundee

1.0 Introduction

The project I have created for this course is a 3D dominoes game. Or, more specifically, it is a variation of the dominoes game called "Muggins" and is played in a 3D environment. The gameplay involves the Player and the Artificial Intelligence taking turns placing domino tiles onto the table, following the rules of the game.

There are two special features that I attempted to tackle within the game. The first is the creation of an interactive and competitive AI opponent. The second feature was an attempt to make the game playable on dual platforms (PC and Playstation2).

1.1 The Gameplay of Dominoes

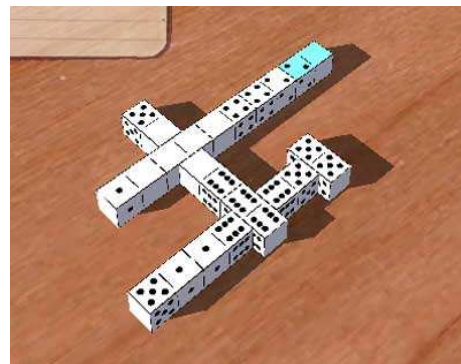
It appears that dominoes was originally a Chinese invention evolving from early dice games, and the domino tile itself represents the combinations that are possible when rolling a pair of die. There are a wide variety of types of domino tiles available in both European and Chinese traditions, as well as countless game types and rules.

Dominoes were common in Europe in the 1800s and experienced a popularity surge in America during the 1920s. I have also recently learned that a form of dominoes shares a place in Eskimo tradition; however I have no knowledge or supporting documentation on this.

The game rules I have implemented are from one of the most popular games called "Muggins", "All Fives", "Five Up", or just "Dominoes" and is played on a standard European six spot tile set.

Each player draws seven domino tiles (or bones) from the stack (or boneyard). The player who drew the highest doubles goes first and may place any domino (from those in their hand) on the table to start the play. The next player must then place a domino that matches one of the ends to the tile played by the first player.

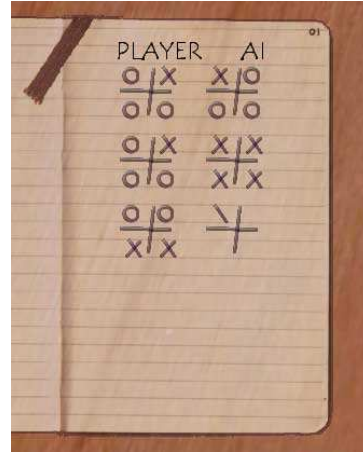
Gameplay continues as the players take turns placing dominoes end to end, however a play only scores points if the sum of the dead ends is a multiple of five. If a player can not match any of the ends, they must draw a 'bone from the boneyard'. If they still can not play, they pass their turn to the other player.



A double domino is placed at 90 degrees and acts as a game branch, and is worth twice as many points until the branch is closed and play continues on the other side of the double. Once closed, the branch stubs do not count for table points, but may be linked. New links to a domino branch are then included in the table sum.

The game continues until one of the players is out of tiles. The other player adds up the face value of their remaining tiles, and the total points (rounded down to the nearest five) are awarded to the player that went out first. The next round then begins like the first, with players drawing seven tiles and the highest doubles determines which player starts.

Game points are recorded in groups of fifty as special marks on a scorecard, called houses. The first ten points of a house are recorded by crossing a vertical and horizontal slash (five points each) forming a large plus sign. Subsequent points are recorded in the corners, with ten points for each quadrant. Symbolically, slashes represent five points and circles represent ten points.

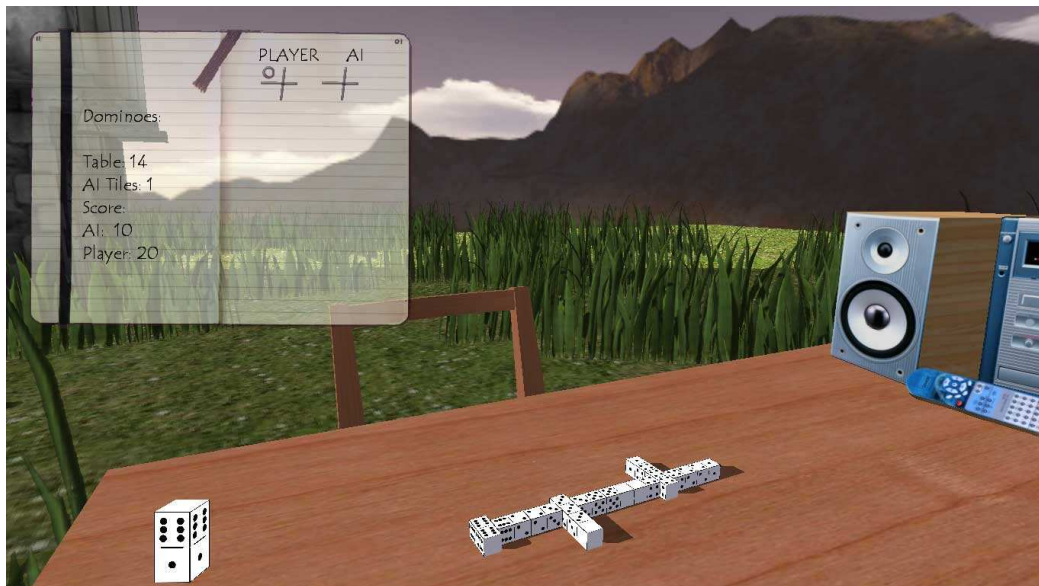


The first player to fill five houses (by scoring 250 points) wins the game.

1.2 The Environment

Although the game can be played on a simple 2D screen, part of the enjoyment of the game is the environment you play in. The environment I have implemented is overly simplified, but I think it shows the potential of what could be accomplished in a bigger project.

What this includes is a basic interactive environment with very limited interaction. There is no physical interaction with the environment, however shadows cast across the courtyard, the grass blows in the wind, and pressing F12 will toggle the radio on and off, providing 3D sound.



1.3 Artificial Intelligence

The original plan was to implement a simple artificial intelligence that would learn from experience. In order to accomplish this, I researched Objective Artificial Intelligence (OAI) as described in "Developing Games that Learn" by Len Dorfman and Narendra K. Ghosh. As I will describe later, I eventually abandoned the idea of using OAI, opting for an effective, although simpler and non-learning approach to the AI.

1.4 PS2 Port

The goal of developing the game for dual platforms provided both challenges and rewards. However, the process proved far more difficult than expected and I believe was only possible because I put PC development on hold, first understanding the limitations of the PS2, and then working the code back to the PC.

1.5 Other Goals

Finally, I think I should mention that I approached this coursework with a few secondary goals in mind. First, I did not want to create another 'FPS', 'Spaceship', or 'Racing Game'. I felt that there was very little for me to gain by attempting to create another game in these genres. For each of these, the gameplay is relatively straight forward and very little can be learned that hasn't already been covered in depth by many others before me. And second, I did not feel that given the resources at my disposal and the time available that I could make a complete game that would stand out. I believe that this is something that I have accomplished by taking the route I chose, at least so far as demonstrating the possibilities of exploring a new spin on a traditional game.

2.0 General Structure

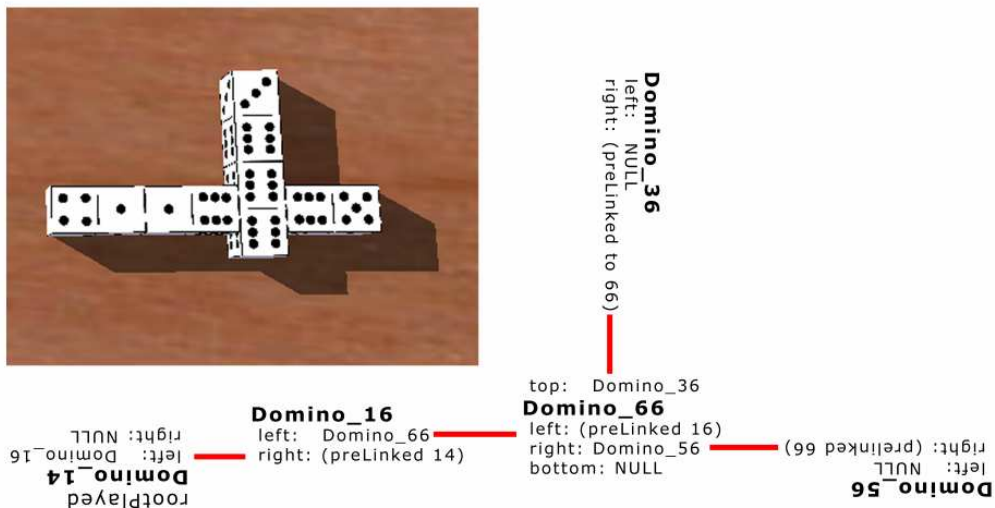
Ignoring the graphical environment for a moment, the structure of the domino game itself proved to be challenging. There are 28 dominoes, and each domino tile is initialized at the beginning of the game as pointers to a domino class.

2.1 The Domino Class

Since there is only ever one domino of any combination (the 1|6 domino is the same as the 6|1 domino) the implemented structure simply ensures that the value of the left side is always less than or equal to the value on the right. Thus the following are the only dominoes are created:

```
0|0  1|1  2|2  3|3  4|4  5|5  6|6
0|1  1|2  2|3  3|4  4|5  5|6
0|2  1|3  2|4  3|5  5|6
0|3  1|4  2|5  3|6
0|4  1|5  2|6
0|5  1|6
0|6
```

When a game player places a domino on the table, it is actually added to a domino tree structure. The top domino or "*rootPlayed*" is the first domino place on the table that round. The *rootPlayed* domino can then have two children (left and right) which must match the domino end values. Double dominoes can have up to four children (left, right, top and bottom). With this in mind, all dominoes (except for the *rootPlayed*) always have one "prelinked" node.



As each domino is linked, it is rotated so that the ends match appropriately. In the figure above, the *rootPlayed* domino is 1|4, and so the child node points to domino 1|6 which must be rotated 180 degrees in order to match the ends. The domino rotation and position is thus derived from its parent and used later when rendering.

With this tree structure, operations and calculations may now take place recursively through the *rootPlayed* Domino, allowing member functions of the domino class to report information back to the program logic.

The dominoes in the player's hand, and the dominoes in the AI's hand are simply a list of domino pointers, as is the *dominoStock* (boneyard), *dominoListOfLinks*. Finally, a *temporaryParentLink* combined with *temporaryLink* flag within the domino class, allows the program to graphically show the player where a link will appear if confirmed.

2.2 Game States

A set of enumerated game states determine what is happening, which screen is displayed, and which keyboard commands work at any point in the program. These should be self explanatory and include: `STATE_PREGAME`, `STATE_GAME`, `STATE_ROUND_FINISHED`, and `STATE_GAME_OVER`.

2.3 Game Functions

Where it was relatively easy to accomplish, game commands that are used by the AI are the same as those used by the player. The idea in allowing the AI and player to share functions was not only to prevent duplicating code, but also to allow for the possibility of interchanging AI and human players (to have AI versus AI or human versus human competitions) as well as in the future having more than two players.

For the sake of meeting deadlines, it was not always possible to prevent the duplication of functionality, however often hooks and/or comments were left in code where the duplicate code could be eliminated in the future (time allowing).

2.4 Porting and Portability

As mentioned previously, additional efforts were made to allow the code to be shared with the Playstation2 implementation of the domino game. The goal was to create a structure that clearly identified the functionality it was attempting to accomplish, so that it could be replaced where necessary.

For example, AI interactions are generalized into enumerated actions (`AI_PASSED`, `AI_PLAYED`, `AI_SCORED`, `AI_SCOLDED`, `AI_HELLO`), which are then separately associated with the audio and visual interfaces. In this way, instead of having a need for the AI logic to interact directly with multiple platforms, the AI simply issues the action, and the actions are then interpreted and executed by the audio and visual interfaces in a way that is appropriate for the particular platform.

3.0 Code Organization

Since the domino game exists within a graphical environment, so too does the domino code exist within the graphical program. Specifically:

- I) MAIN ENVIROMENT
 - a. The Domino Game
 - i. Game Play
 - 1. AI Actions on Dominoes
 - 2. Player Actions on Dominoes
 - ii. Game Audio Interface
 - iii. Game Graphical User Interface
 - iv. Game Keyboard Controls
 - v. The Dominoes
 - 1. The Domino 3D Meshes
 - 2. Self-Recursive Functions
 - b. The Mouse Interface
 - c. The Camera
 - i. Camera Keyboard Controls
 - d. The Lights
 - e. The Graphical Options
 - i. Graphical Options Menu Display
 - ii. Graphical Options Keyboard Controls
 - f. The Environmental Meshes
 - g. Windows Interface

3.1 General Coding Style

In most cases, each program unit is a class, and each class is given its own .cpp file and associated .h header. And while references to the domino class are via pointers, most other game interaction with classes is through explicit declarations.

I also chose to use explicit init() and cleanup() functions for each class rather than relying on constructors and destructors. Rarely is a function overloaded, instead it is more often that a default parameter is used. For example, in my code from last semester, the staticMesh class had member functions draw() and drawShadows(). As I moved forward in consolidating my code there is now only one member function draw() with a Boolean "render as shadow" parameter that defaults to false.

Also, each time a class needs access to the Device, it is declared as a member pointer within the class and passed to the init function as a parameter and is set immediately. Through these types of standard practices I attempted to keep control on code base that grew rapidly.

Additionally, I developed some better coding practices as I moved forward. Advised by some of the students from Turkey, I started to prefix parameter variable names with 'p_' and class member variable names with 'm_'. With this in mind, it becomes easy to pick out which parts of my code are older than others.

I also made an attempt to keep the majority of the class member functions and variables as private, only making them public when necessary. I was significantly less successful at this than I had hoped, often for the sake of speed, making variables publicly accessible rather than writing a function to interface to them.

3.2 Noteworthy Classes

3.2.1 Class - *domino*

By far my most significant structural achievement within this program, the domino class had to be given considerable thought and suffered a few false starts during its creation. At one point I was faced with necessary game functionality that I was convinced would require me to rewrite the entire structure. Fortunately, like many problems, the answer was to walk away for 24 hours and the solution presented itself, preventing me from starting over.

As discussed earlier, the domino class consists of four child nodes (pointers to the domino class definition). The face value of the left and right side of the domino are stored as variables and set on initialization.

I used the member function *render2d* to test the class functionality before any 3d interface, later adding the *render3d*, *render3dS* (for shadows), and *renderGUI* (for dominoes in the player's hand). Other than *renderGUI*, the other three render functions are recursive.

However, while the logic behind the member function *calculate_values()* is enjoyable to wrap your brain around, by far, the most mind-numbing and most important logic within the whole code structure is the function *link()*.

What *link()* does, is to determine which child links within the tree structure are available nodes, how the new domino should link to that node (rotation), and what the new offset position for the child domino will be. If a node is flagged as 'temporary', a pointer to the node parent node is recorded so that the temporary link can later be confirmed or removed.

3.2.2 Class - *dominoGame*

The dominoGame class is probably the class with greatest scope, meaning it has more unrelated items than any of the other classes. It not only holds the majority of the game logic, but also (unfortunately) the code for the AI, the "global" game variables (like game states and game scores), and the references to the dominoes discussed above.

I am disappointed that I did not have time to separate out some of the items listed above into their own classes, because I'm sure if I did, it would make it easier to add new domino game types in the future.

Trying to bring some order to this chaos, I have divided the member functions into five categories, and attempted to use function names that clearly represent their purpose:

```
//FUNCTIONS: Game Init and Cleanup
bool  init(IDirect3DDevice9 *);
void  createDominoes(void);
void  initDominoes(void);
void  Cleanup(void);
void  deleteDominoes(void);
int   startNewGame(void);
int   startNewRound(void);
void  setPlayer(playerType);

//FUNCTIONS: Domino and Domino List Manipulation
void  bubbleSortDominoesInPlayersHand(void);
void  check_links (CDomino *ptr, int left, int right);
void  clearDominoListOfLinks(void);
void  filldominoListOfLinks(void);
void  resetDominoStock (void);
void  shuffleDominoStock(void);

//FUNCTIONS: Player/AI initiated actions
void  changeDominoInPlayersHand(int);
void  changeLink(int);
void  confirmLink(void);
void  drawTile(playerType);
CDomino *selectFromDominoStock (void);

//FUNCTIONS: Game Displays
void  renderScene(void);
void  renderSceneShadows(D3DXMATRIX shadowM);
void  renderPlayerDominoes(void);
void  renderPreGameScreen(void);
void  renderScoreboard(void);
void  renderTitleScreenText(void);
void  renderRoundFinishedScoreboard(void);
void  renderGameOverScoreboard(void);

//FUNCTIONS: General Purpose
void  playAI (void);
void  awardPointsToCurrentPlayer(void);
void  checkKeyboard(float);
```

3.2.3 Classes – *gameGUI* and *gameAudio*

These two classes are both referenced directly by the *dominoGame* class, and as mentioned earlier, provide an outlet for messages from the AI. It is the visual and auditory manifestation of these messages that makes the domino game actually feel like a game. Without them, the game is much less fun because it is difficult to understand what is happening without constant cues and updates from the game.

For example, with the *AIThinking* flag and associated GUI message, the game *felt* like a computer program. The responses were fast, efficient, accurate, and extremely boring. Simply by adding the message “AI is Thinking” for an arbitrary number of frames, it immediately became a more engaging experience by adding a factor of time. The inverse was equally successful (having the AI yawn if it is taking a long time for the player to move.)

I should mention that although modified to address my game requirements and coding style, a significant amount of the gameAudio logic came from [3D Sound with DirectX Audio](#) by Toby Murray. This includes the CDXAudio and C3DSound audio classes.

3.2.3 Noteworthy Code Contributions

Other significant code contributions include the gameMouse code and the wavyGrass effects. The gameMouse code was pieced together from the tutorial [Developing a GUI Using C++ and DirectX – Part I](#) by Mason McCuskey.

The wavyGrass class and associated .fx file came from [HLSL Cg Demos and Tutorials – Using Vertex and Pixel Shaders \(Effects Files fx\)](#), which provided an actual Visual Studio project file to view and test the effects.

I should also mention that at a particularly rough point, Kaya Oguz provided an extra set of eyes which helped lead me to finding an unreferenced pointer crash that had previously eluded me.

3.2.4 Code Legacy

Finally, I would note that this code is a progression of my work from first semester. The core code logic (backed by first semester's examples) has remained relatively unchained with a few exceptions.

Most notably, I have updated the way the staticMesh class renders to the screen. The staticMesh class is used to load .X meshes. Not only have I consolidated the draw() and drawShadow() functions, but I have also converted them to use my effects file, which itself has roots from the skinnedMesh effects file of last semester. The effects file now has three techniques: VBlend2Tech, VBlend2ShadowTech, and VBlend2SkyboxTech.

And since I have simply expanded on the previous effects file (although none of the meshes currently in the game are animated) by simply making use of my skinnedMesh class from last term, animated meshes can easily be incorporated with the current framework without major modifications.

4.0 Special Features and Critical Evaluation of Special Features

As described earlier, there are two special features that I wanted to implement. The first and arguably more relevant to making game was the desire to create an interactive and competitive AI opponent. The second feature, which may be slightly easier to objectively gauge the success, was to make the game available on dual platforms.

4.1 Special Feature – Artificially Intelligent Opponent

In order to make the AI both interactive and competitive, it seemed appropriate to explore techniques that allow an AI to learn as it plays, which led me to Objective Artificial Intelligence (OAI) as described in Developing Games That Learn. The OAI architecture developed by the authors involves AI learning through the old adage, "Fool me once, shame on you; fool me twice, shame on me."

The basic idea is to think of the coded AI as "instinct". When the PC loses, it records the series of moves made that led to the loss (generalizing the pattern for things like symmetry) and stores this to disk. It would also record the resulting pattern of "wins". This then becomes the AI's "memory". During the next match, before making a move, the program first does a quick check of memory to see if this pattern has occurred previously (either as a win or loss), and makes a move to prevent the loss pattern or to repeat a winning pattern. If the pattern is not in memory, the program returns to the instinct to decide which move to make.

The book gives a set of questions to analyze whether or not the OAI technique is useful for your code (game or otherwise). The games of "Tic-Tac-Toe" and "Connect-Four" are provided in the book as demonstrations of AI learning through the OAI technique.

On initial inspection this appeared to be very appropriate, since like the game of dominoes, the sample games are both traditional board-games that can be played with tiles and follow a simple and specific set of rules. However, after deeper analysis I was not so sure about the value of applying this technique.

The problem is that the game of dominoes is different than the example games in a couple significant ways.

First, players in Tic-Tac-Toe and Connect-Four only have one type of move to make, and although both games have two types of tiles (an 'X or an O' and a 'Black or Red chip'), each player is only represented by one of these. The tile types only serve to distinguish the players. In Dominoes there are 28 different types of tiles that the player might play, and they are not player specific.

Secondly, in the example games, the player is not limited by the number of pieces in their hand. They can play as many chips as they want until the game is over. This means that the code can be written to perform the

same logic each time, only analyzing the patterns on the board. However, In Dominoes the choice and number of tiles is different every round, so at a minimum, the code has to look at both the game board AND the tiles in hand to determine if there is a repeated pattern.

Again, on analysis of the game of dominoes, I could only think of a few limited situations where a computer might learn not to make the same mistake twice, but it seems much more important to have the AI just make sure that they are not "setting the player up" on the subsequent move. However, this would need to be a weighted decision, because a "bad move" might turn out to actually be the best move in the given situation.

Further, in Dominoes, since the game topology is different based on the tiles previously played and the way in which they are linked, the number of types of patterns are significantly higher. I think this would require playing the games hundreds of times before the AI was faced with a similar situation which it would have learned from.

In the end, my research led me to believe that in dominoes, what the authors call 'instinct' is far more important than 'memory'. Or, to put it in another way, success in Dominoes is not achieved by memorizing patterns. Success is instead achieved by analyzing the current situation and making the best possible move each time, given the current set of circumstances. It is not often that one move forces the other player to make a second specific move, because the player is rarely aware of the other player's dominoes.

In my analysis, the game of dominoes began to appear more like a series of mini-games that occur each turn, where it is rare for the decisions made this turn to have a direct impact on the decision that must be made the next. Each individual decision to score indirectly leads to success in the game as a whole.

With this in mind, my first plan for Artificial Intelligence was to create three levels of difficulty with the following logic:

Easy: The AI places the first scoring tile in finds. If none are found, it places the tile with the highest value total value.

Medium: The AI places the tile that scores the highest points for that round. If no tile will score (or if multiple placements result in the same score), the AI places the tile with the highest point value.

Hard: Same as Medium, however the AI also checks the possible combinations of tiles that remain available for the other player to use. If the analysis led to a situation that would leave it open for the player to follow up with a significantly high score, the AI would not make the play unless necessary. (A weighted value is given to each possible move, and the AI chooses which move to make accordingly.)

Additionally on the 'Hard' setting, the AI should 'remember' if the player had the first move, which would mean the player had one of the double tiles. (This is one of the few situations in which the player knows for sure the other player's tiles.) In this case, the AI knows that the player has the double tile and would use that when weighing the decision.

The actual AI I programmed involved a combination of the Easy and Medium levels described above, and immediately proved to be challenging enough for most players. In fact, during the first round I played against the AI, I found myself losing by 30 points within the first minute.

I have held off developing a hard version of the AI that involves a weighted analysis of probability because game testing revealed something more important. I soon discovered that although I had developed a competitive AI, I had not yet addressed the other original goal of my AI design, specifically "interactivity".

Interactivity was based less on research, and more from personal experience. I knew I wanted the AI to provide an auditory response, but one that did not seem repetitive. I knew that I wanted the AI be a 'pleasant' opponent and thus demonstrate happiness when scoring.

These 'personal touches', became as important to the gameplay as the logic and I believe demonstrate that computer games are not strictly about hard-science problem solving. There is definitely room to explore implementing knowledge from the fields of psychology and sociology.

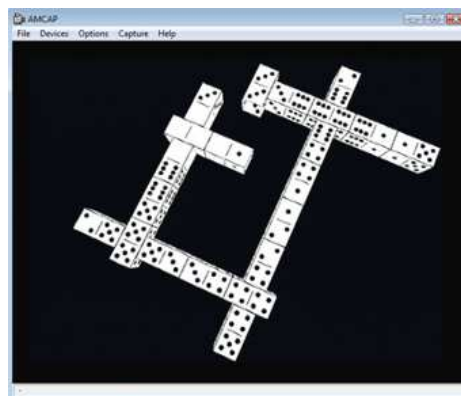
On a final note, at some point in the process I realized that the game of dominoes is much more like a card game than a board game. I'm not sure why this was not more obvious from the beginning, but I now believe I would have benefited from researching AI techniques used in these types of games of chance.

4.2 Special Feature – Porting to the PlayStation2

I have already briefly touched on the experience of porting the game to the PS2. Research was primarily focused on understanding the differences between the two systems and finding a way to not pigeon-hole the code into one platform.

The flexibility of DirectX compared to the rigidity of the PS2 platform led to the obvious solution of first developing the core code on the PS2, and then porting that code back to the PC.

For simplicity I did not attempt to port the environment, but instead focused on the actual game of dominoes (as you can see in the adjacent screen shot.)



The gameplay is very similar on the PS2, since the logic and audio and visual cues performed by the AI are identical. Also, since the control scheme was first developed on the game controller, it was easier to convert to the keyboard, than if I had developed on the keyboard and tried to map keys to the game controller.

5.0 Critical Appraisal

There are a variety of things about my code that I am not happy about, and although my understanding of the fundamentals of game programming with DirectX have increased significantly since my previous semester's project, there are still some areas within programming design patterns that I am unsure about.

5.1 Problem Areas in Code Organization

There are three areas of the program that I believe would benefit from a significant overhaul. The first, as describe earlier, would be to partition the dominoGame class into areas with a more focused scope.

The second major portion of code that I feel needs to be addressed is the main.cpp. I do not like that the individual meshes are handled in the main loop. Given more time I would create a class to manage all game environmental game meshes.

And finally, I am not happy with where the AI logic ended residing, and how it interacts with the rest of the game code. I would have much preferred to be separate the AI into its own class and have the game interface with the AI the same way the game interfaces with the player. This was not as critical for this project, but would be important in expanding the AI functionality in the future.

I also don't like that the AI logic occurs in one frame. I would have liked to explore spreading the AI decision making across multiple frames. However, there is one situation when this does occur. Specifically, when an AI can not play and must draw a tile, rather than deciding what to do with the new domino, the tile is simply added to the AI's hand and the AI logic starts over on the next frame.

5.2 Where to Put the Interface

Again comparing to my work of the prior semester, I believe I was more successful at designing an appropriate class hierarchy. For example, there are multiple interfaces to the keyboard, each specific to the function of the individual class. Also, the decision to separate the graphical environment from the primary gameplay proved to be beneficial.

However, I found I was constantly revisiting the wisdom of these architecture decisions, and while they worked well, I still feel that I have not designed the 'perfect' framework. Although I do believe I am moving in the right direction.

5.3 Graphical Effects Feature

I added the wavyGrass class as an environmental effect later in the game. It uses a vertex shader to move the top of the grass blades around in a nice effect. However, there are two things that stand to me as potential performance problems.

First, I believe that using sin and cos functions is costly, and perhaps they should be replaced by a faster trig lookup table. And second, I wonder if it would be more appropriate to move those same calculations into the HLSL code and then share the calculations across multiple nodes instead of making a separate calculation for every node.

5.4 Project Assignment – Collision Detection

Finally, on reviewing the project requirements, there is one area that I did not address. There was no need within the gameplay to have conventional collision detection, since the physical interaction of the individual dominoes is controlled by the recursive linking implemented in the domino class.

The outside 3D environment however would have benefited from implementing basic collision detection; however it was just as easy and appropriate within the objective of the gameplay to lock the player from moving within the environment. As such, the player is then prevented from “zooming in” on the game board by just moving closer.

Given more time, I would have attempted to address the zoom issue by modifying the projection matrix. Although, it did not make it on my critical path, the plan was to use the mouse scroll wheel to modify the projection matrix, thereby increasing the magnification.

Conclusions

I find it extremely difficult to put down the keyboard. It seems like there is always another bug to fix or another feature to add that will make the game significantly better. I'm disappointed that I have not had time to add many of the little things that didn't make it onto my critical path list. (For example, animated seagulls that soar overhead and squawk in the distance, a more interactive physical environment, and the sounds of the wind through the grass.)

With that said however, I am very happy with what I have been able to accomplish with the game. I and my game testers have found it fun to play and fun to interact with. Based on this, I think it is something that I will continue to work on and improve.

I am also very pleased with the graphical representation of the scorecard. This was more difficult than planned, but produced a nice graphical effect.

As with any project of this magnitude, you can't help but learn a great deal. I feel much more confident in both C++ as well as in understanding the framework behind DirectX functionality. I am finally seeing a connection between the practices in dealing with the different product families (D3D, DirectInput, and DirectAudio).

I remember writing last term that it seemed like it would be advantageous to use .fx files for all modelling effects and move away from setting render states completely. An interesting and unfortunate side effect of using the effects framework is that I found that it was easy to make a small logic mistake that the compiler did not catch, but at runtime, everything would disappear. Those bugs were hard to diagnose because they were often hidden in the interaction between the C++ and the HLSL.

Also, in porting the program to the PlayStation2, my initial reaction was a feeling that things were much easier when you didn't have a constantly changing operating system getting in the way of game processing. However, on returning to DirectX I was amazed at how much more I could accomplish in a limited amount of time thanks to the robust library of capabilities that come wrapped into DirectX.

The best piece of advice on improving the gameplay that I would give someone who is attempting a similar task is to keep testing. Get as many friends to give your game a try as soon as you have it "kind of" working. My greatest knowledge gains about how to proceed occurred when I watched someone else try to play.

But my best piece of advice on the actual programming is that the only way you're ever going to actually learn is to just "start coding".

References:

Dorman, Len (1996), 'Developing Games that Learn', *Prentice Hall PTR*

Fortuna, Dr. Henry (2008), Course Lectures and Notes, University of Abertay Dundee

HLSL Cg Demos and Tutorials – Using Vertex and Pixel Shaders (2008), viewed 1 May 2008, <http://www.xbdev.net/shaderx/fx/index.php>

McCuskey, Mason (2008), Developing a GUI Using C++ and Direct X – Part I, viewed 1 May 2008, <http://www.gamedev.net/reference/articles/article994.asp>

Natanson, Dr. Louis (2007-2008), Course Lectures and Notes, University of Abertay Dundee

Oguz, Kaya (2008), Help with Debugging, University of Aberay Dundee