Design Specification

Introduction

Goals and Objectives

GameForge is a graphical tool used to aid in the design and creation of video games. It attempts to bring game development down to a level that any computer savvy user can understand, without requiring masterful programming ability. A user with limited Microsoft DirectX and/or Visual C++ programming knowledge will be able to construct a basic, 2-D arcade game. GameForge limits the amount of actual code written by the user, if not eliminating it completely. It will also assist experienced programmers in generating the Microsoft DirectX and Microsoft Windows9x overhead necessary for basic game construction, allowing them to concentrate on more detailed game design issues and implementation.

Project Scope

GameForge is a graphical tool used to aid in the design and creation of video games. A user with limited Microsoft DirectX and/or Visual C++ programming knowledge will be able to construct a basic 2D-arcade game. The idea is to limit the amount of actual code written by the user. It will also assist experienced programmers in generating the Microsoft DirectX and Microsoft Windows9x overhead necessary for basic game construction, allowing them to concentrate on more detailed game design issues and implementation.

The software will consist of a number of inputs, graphically assisting the user in creating on-screen objects including the following:

- User Created Objects (player character, creatures, static objects)
  - Bitmaps (with animation)
  - Collision Detection Areas
  - Movement Routines
  - Additional Object Attributes
- Backgrounds
- Input Device Setup
- Sound Events

The software will also consist of a number of graphical processing functionalities including the following:

- Defining/Editing Objects (including characteristics)
- Object Positioning
- Opening/Closing/Saving Game Project Files
- Exporting Game Projects to compilable C++ Files

Outputs include:

- User Created Sprite Objects
- Bitmaps
- Microsoft VC++ (with DirectX code) Files
- Game Project Files
- Text Files (containing sprite attributes)
- Database Files

Software Context

GameForge is being marketed as a CASE tool, to allow software developers to 'build' rather than code their game. It is not necessary for developers to have prior knowledge with DirectX or Visual C++, as long as they have a good art team and high production values. GameForge will be commercially distributed via the GameForge website (for information regarding the URL, see the Appendix.)

GameForge will be available free for educational use. It will be distributed for use in CIS 587, Computer Game Design and Implementation, at the University of Michigan-Dearborn.

Major Constraints

Performance/Behavior Issues

GameForge is designed to be compatible with the Microsoft Windows 9x operating system. Microsoft Windows NT 4.0 and earlier versions will not be supported (Windows NT only supports Microsoft DirectX up to version 3.0. DirectInput had not been implemented at this time, making this version of DirectX very limited.) Microsoft Windows 2000 should also be compatible.

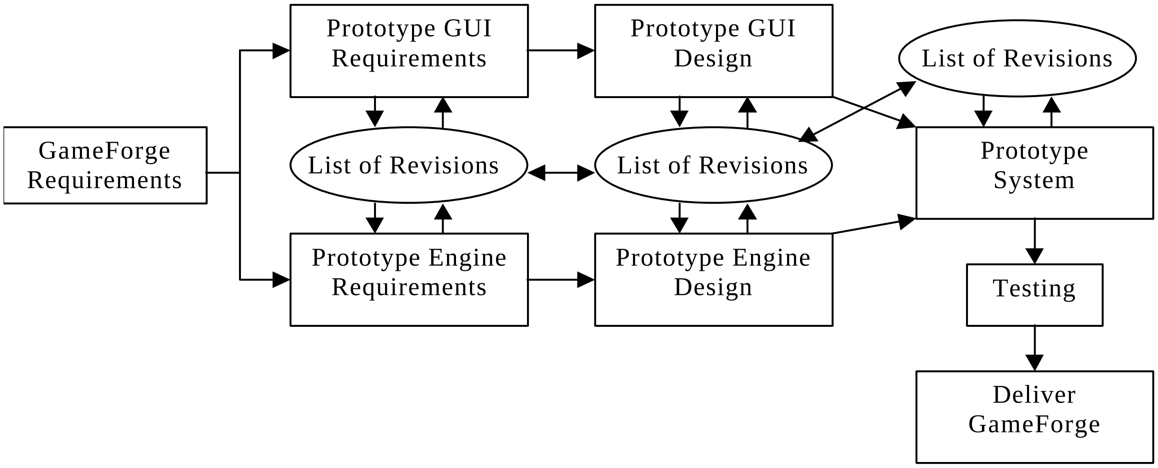
GameForge also requires Microsoft DirectX 7.0 or above. Users may also want to obtain the DirectX 7.0 SDK if they plan on expanding the GameForge library files beyond their original scope.

GameForge also requires the Microsoft Visual C++ 6.0 compiler. GameForge's VC++ code may be compilable using Borland or some other VC++ compiler, but functionality is not guaranteed.

Management and Technical Constraints

GameForge has a drop-dead delivery date of 04/17/00.

PA Software will be using the Rapid Prototyping model during design and implementation:



Data Design

Internal Software Data Structures

Sprites:

Sprites consist of the following attributes:

Name (Primary Key) – Name of the sprite.
Image – Name of the Image file displayed representing the sprite.
Width – The width of the sprite in pixels.
Height – The height of the sprite in pixels.
DestinationX – The X coordinate for the destination of the placement of the sprite.
DestinationY – The Y coordinate for the destination of the placement of the sprite.
Framerate – The framerate of the sprite.
NumOfDir – The number of directions the sprite has.
NumOfFrames – The number of frames per direction.
Solid – Whether or not the sprite is a solid object.
KillsPlayer – Whether or not the sprite kills the player.
PlayerCanKill – Whether or not the player can kill this sprite.
OtherCanKill – Whether or not other sprites (other than the player) can kill this sprite.
Obtainable – Whether or not the sprite can be picked up.
Visible – Whether or not the sprite is visible on the screen.
AffectsScore – Whether or not the sprite has an effect on the score.
SoundAttached - Index of the sound is attached to this sprite.
ReactsToGravity – Whether or not the sprite is affected by gravity.
ReactsToFriction – Whether or not the sprite is affected by Friction.
ReactionToPlayer - Index determining how the sprite reacts to the player’s position.
Bounces – Whether or not the sprite changes direction when contacting another sprite.
Random – Whether or not the sprite has random movement.

Surfaces:

Surface consist of the following attributes:

Alias (Primary Key) – The filename of the image.
Path – The directory that the image is found in.
Height – The height of the image in pixels.
Width – The width of the image in pixels.

Messages:

Messages consist of the following attributes:

String (Primary Key) – The actual text to be displayed.
DestX - The X coordinate for the destination of where the string will be placed.
DestY – The Y coordinate for the destination of where the string will be placed.
ForeRGB – The RGB value of the foreground color of the text.
BackRGB – The RGB value of the background color of the text.
Transparent – The background of the text can be made transparent.
Visible – Whether or not the text is visible on the screen.

Sounds:

Sounds consist of the following attributes:

Alias (Primary Key) – The filename of the sound.
Path – The directory that the sound is found in.
Buffer – The buffer to load the sound onto.
Notifications – Breaks up sound into pieces for streaming.
State – Determines play state (playing, not playing, looping).

Levels:

Levels consist of the following attributes:

Name (Primary Key) **–** The name of the level.
Width Overall width of the level (in pixels).
Height Overall height of the level (in pixels).
GoalSprite Index of the goal sprite.
GoalScore Value of the goal score.

Global Data Structures

Object Handler:

Object Handler is the only global structure. It contains the following:

Object Array – Array of non-moving sprites.
Creature Array – Array of moving sprites.
Background Array – Array of background sprites.
Player – Player Sprite.
Message Array – Array of text messages.
Surface Array – Array of surfaces.

Game – Contains game information.
Logic – Contains all game logic.
DirectSound – Handles all sound logic.
DirectDraw - Handles all drawing logic.

Temporary Data Structures

No temporary data structures are created.

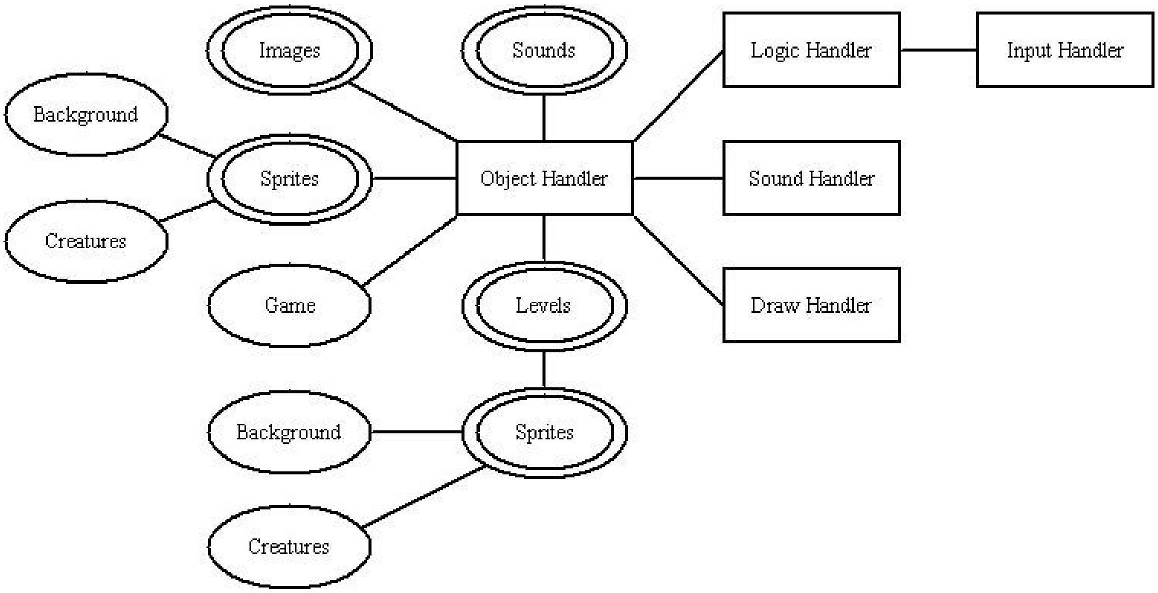
Database Description

The database we are using is a Microsoft Access Database. It will be created using Microsoft Access 97. It will hold all of the information entered by the user during the design of the game. This information will be stored in the tables that are listed above in the section titled **Internal Software Data Structures**.

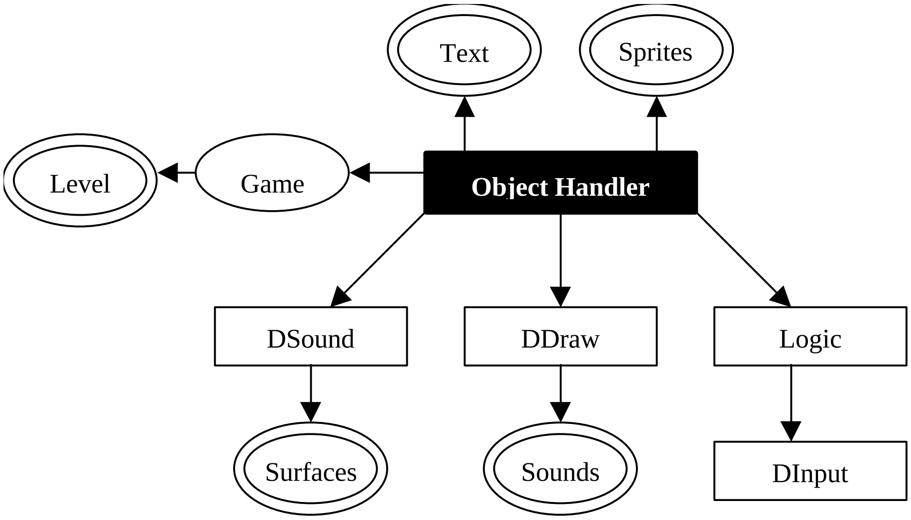
Architectural and Component-Level Design

Program Structure

Architecture Diagram



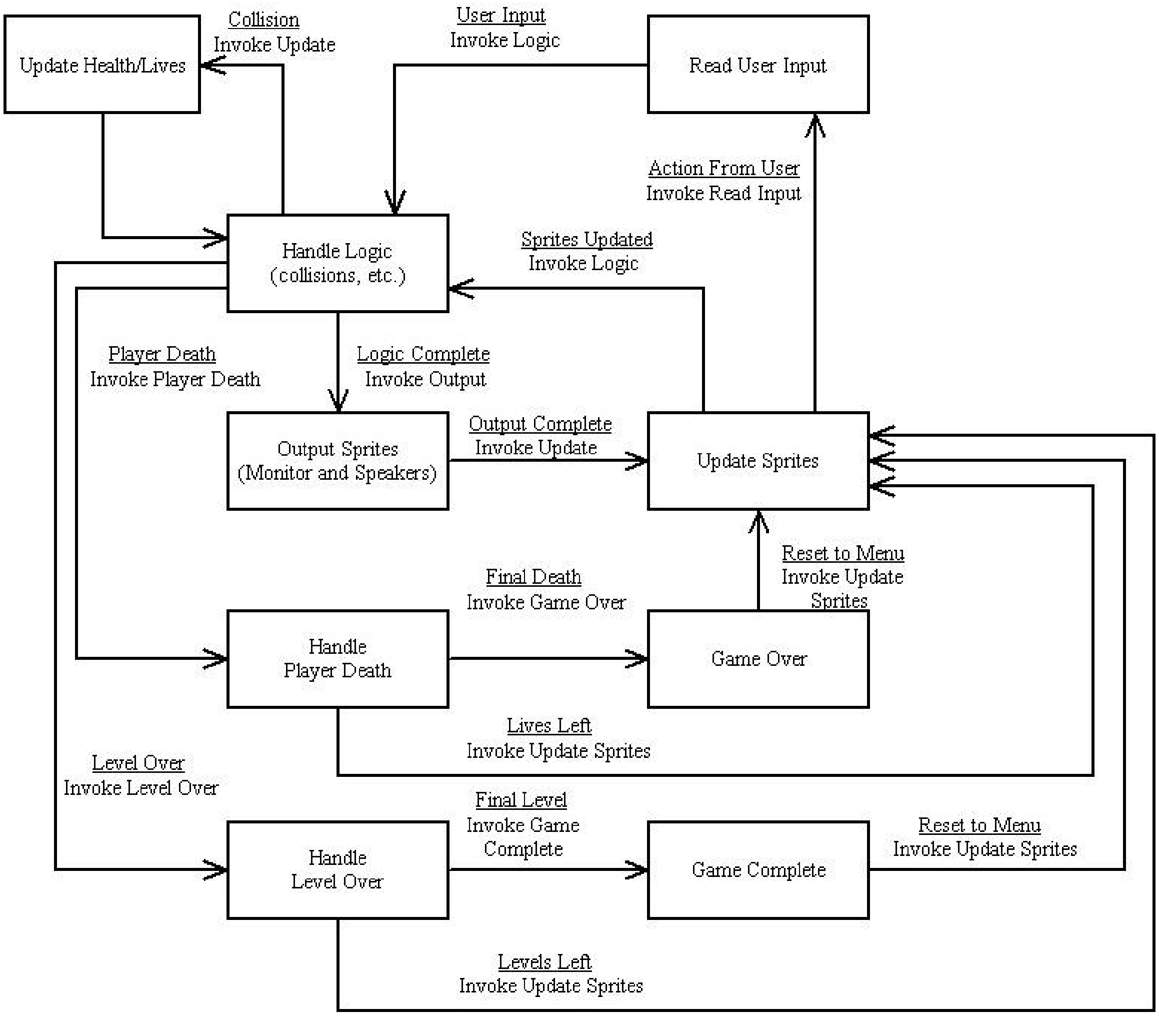
Alternative Architecture:



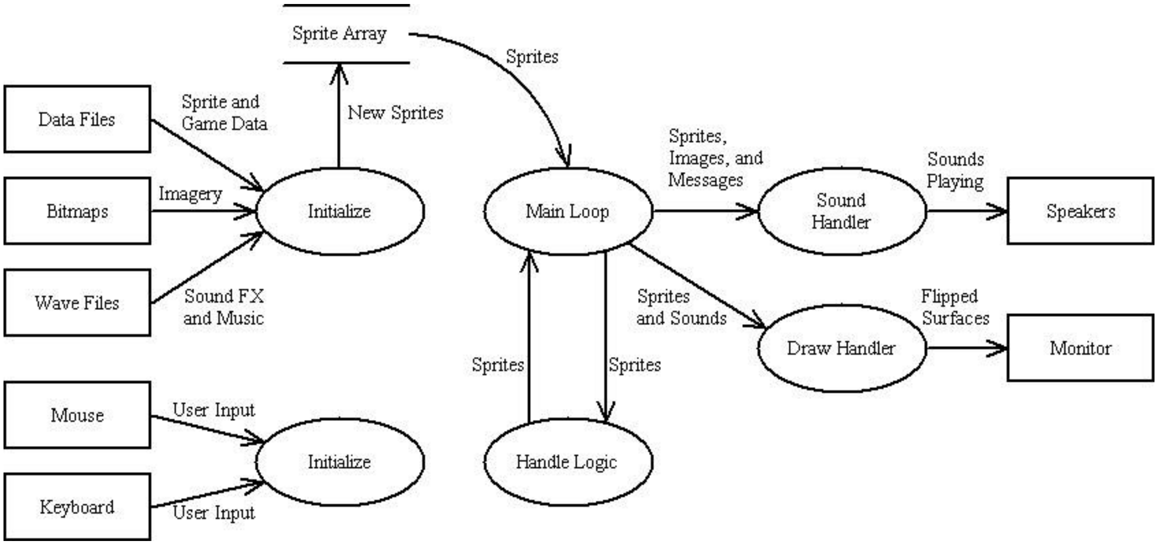
It is important to note that the proper engine architecture breaks up the Sprite structure into a number of categories, allowing each of these categories to communicate directly with the Object Handler. Without this breakdown, a significant amount of unnecessary data must be passed back and forth through the Object Handler to parse down the Sprite information. With the current design, the Object Handler can easily retrieve Sprite information after receiving data requests from the other Handlers, with a minimum amount of parsing.

It is also important to note that DDraw, DSound, and DInput each require surfaces, buffers, and objects, respectively. By removing these structures from their subsequent functions, we can increase modularity, and allow easy updating of these structures, without affecting their functions.

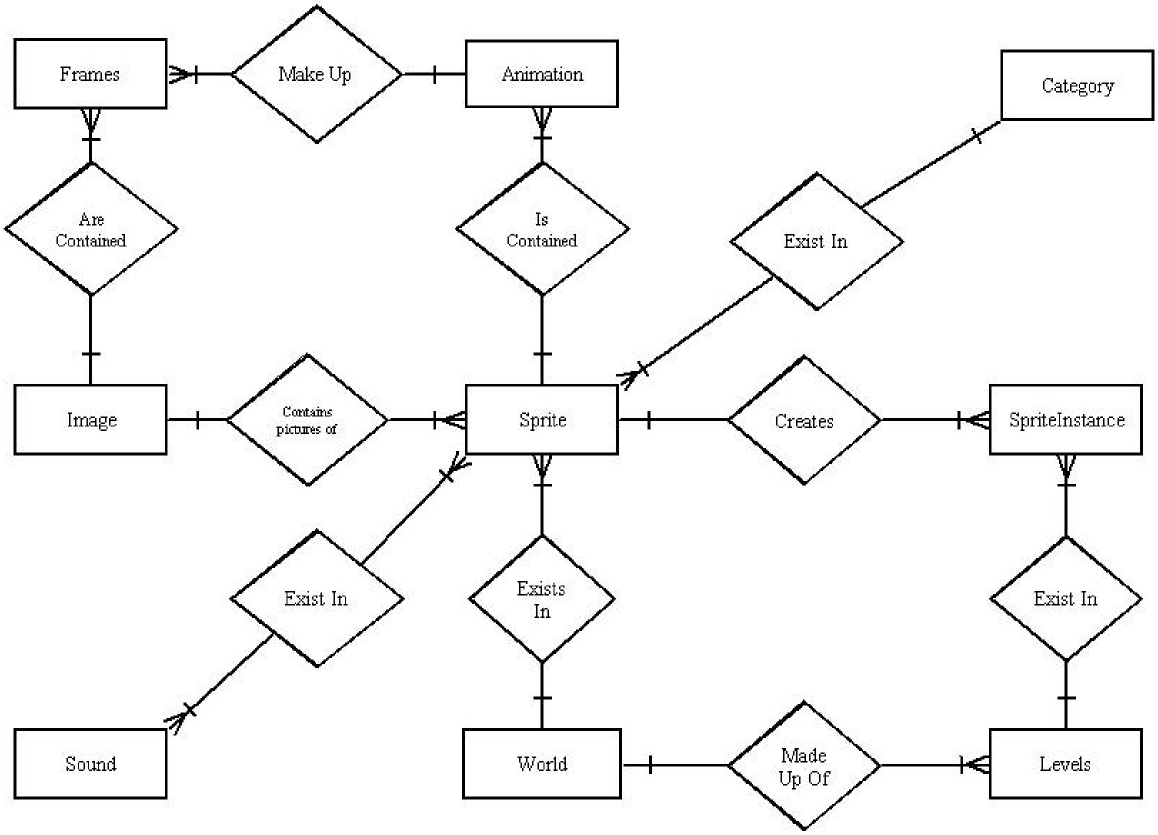
Engine State Transition Diagram:



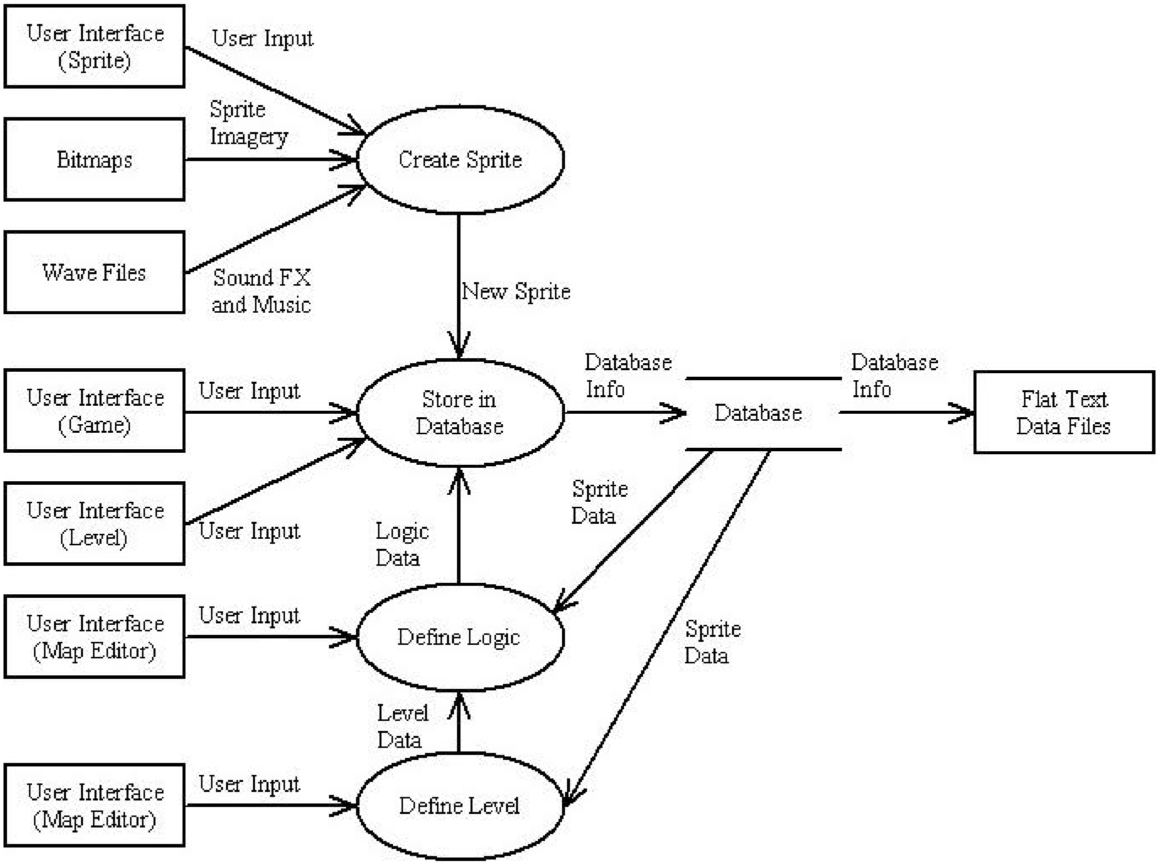
Engine Data Flow Diagram



World Builder Entity-Relationship Diagram (ERD)



World Builder Data Flow Diagram



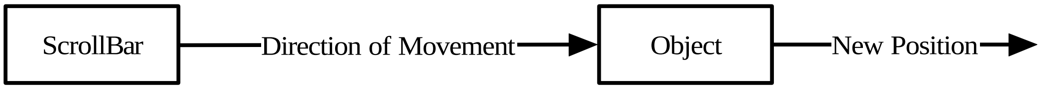
Component Descriptions

INTERFACE

Scrollbar_Change:

Narrative: Receives no input. It's job is to scroll the object that it is attached to in a the direction the user chooses. Any time the user clicks on the scrollbar, this function handles how the user is interacting with the scrollbar. Scrolling up moves the data up, scrolling down moves the data down. This is done by interpreting how the user is interacting with the scrollbar and changing the top and left coordinates of the image by some pixel value (depending on how much the user moves the bar)

Diagram:



Interface: The function interfaces with the object that the scrollbar is linked to. It changes the way the information is displayed in that particular object.

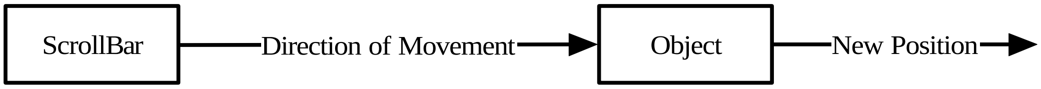
Issues: If the function is not handled correctly the data in the object linked to the scrollbar may not be displayed in the manner intended.

Constraints: All scrollbars must be associated with the proper object(s) or the proper output may not be displayed.

Scrollbar_Scroll:

Narrative: Receives no input. It's job is to scroll the object that it is attached to in a the direction the user chooses. Any time the user clicks on the scrollbar, this function handles how the user is interacting with the scrollbar. Scrolling up moves the data up, scrolling down moves the data down. This is done by interpreting how the user is interacting with the scrollbar and changing the top and left coordinates of the image by some pixel value (depending on how much the user moves the bar)

Diagram:



Interface: The function interfaces with the object that the scrollbar is linked to. It changes the way the information is displayed in that particular object.

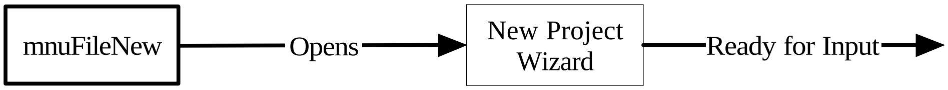
Issues: If the function is not handled correctly the data in the object linked to the scrollbar may not be displayed in the manner intended.

Constraints: All scrollbars must be associated with the proper object(s) or the proper output may not be displayed.

MnuFileNew_Click:

Narrative: Receives no input. It's job is to load the correct interface (wizard) to allow the user to create a new project. The user will then be able to use this new interface to input data that the program will use to create the final product.

Diagram:



Interface: The function interfaces with the wizard that will prompt the user for the information needed to produce the final output of the program.

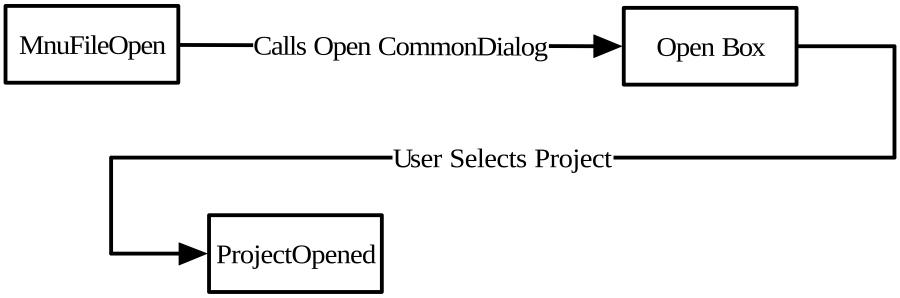
Issues: The user should be ready to lose all information that GameForge currently has open.

Constraints: All data currently open will be lost if the user is not prompted to save current open data, or if the saving is not done automatically.

MnuFileOpen_Click:

Narrative: Receives no input. Its job is to load the correct project files into memory that the user chooses. The user will then be able to work with the project.

Diagram:



Interface: The function interfaces with the common dialog control that is the standard "Open" box associated with most Microsoft Windows applications. The user will choose a file displayed in this control.

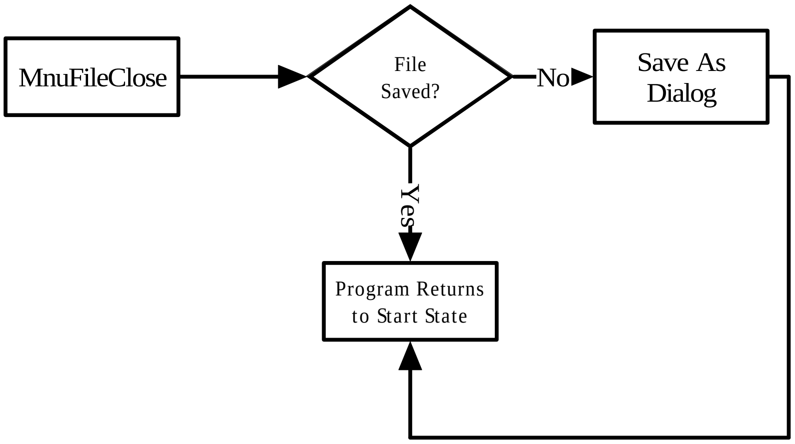
Issues: The file chosen must be a data file that is in a format supported by GameForge.

Constraints: Without the common dialog control the user will not be able to choose the file to open. This is included in the application wizard.

MnuFileClose_Click:

Narrative: Receives no input. Its job is to close all files. Any cosmetic “closing” effects are also included here.

Diagram:



Interface: The function interfaces with the common dialog control that is the standard “Close” choice associated with most Microsoft Windows applications.

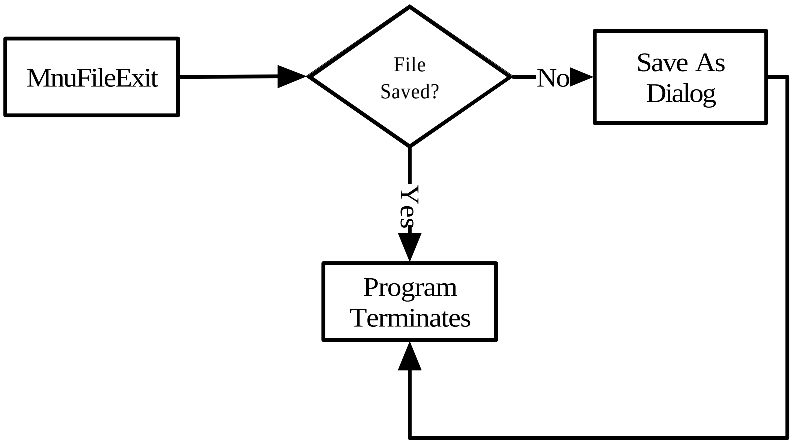
Issues: None.

Constraints: Without the common dialog control the user will not be able to choose this command. This is included in the application wizard. The user should be prompted to save their files to disk or else they will be closed and changes will not be saved.

MenuFileExit_Click:

Narrative: Receives no input. Its job is to close all files, and exit the program. Any cosmetic "closing" effects are also included here, and the main program is unloaded from memory.

Diagram:



Interface: This function will need to interface with the save as dialog. Because if it doesn't the user's files will be closed and changes will not be saved.

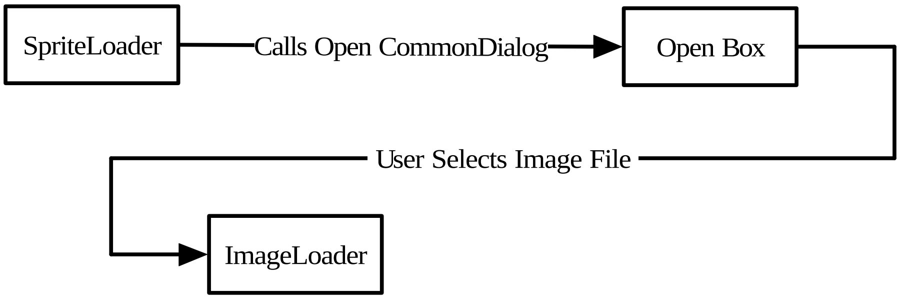
Issues: The user should be prompted to save their files to disk or else they will be closed and changes will not be saved. The user should be asked to confirm the desire to exit the program.

Constraints: If this function interfaces with the SaveAs Function, the Microsoft Common Dialog controls will need to be included.

SpriteLoader:

Narrative: Receives no input. Its job is to pass the filename of the sprite into the image loader.

Diagram:



Interface: The function interfaces with the ImageLoader function, and the "Open" common dialog. Imageloader is the function that loads the images into a particular object.

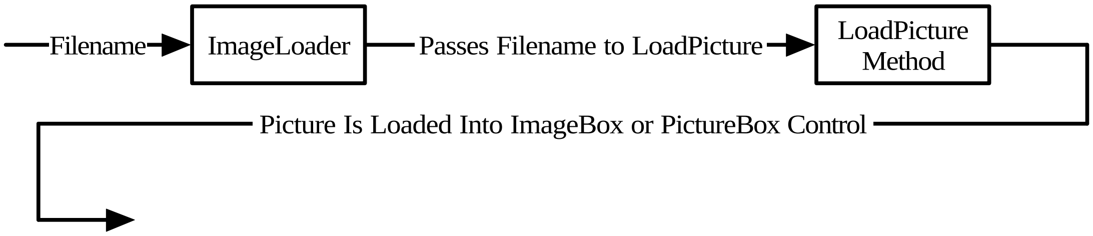
Issues: The file chosen must be a bitmap.

Constraints: Without the ImageLoader function and the common dialog control the user will not be able to choose the file to open. The common dialog control is included in the Visual Basic application wizard.

ImageLoader:

Narrative: Receives the filename to display as input. Its job is to load the correct sprite on the screen.

Diagram:


Filename
ImageLoader
Passes Filename to LoadPicture
LoadPicture Method
Picture Is Loaded Into ImageBox or PictureBox Control

Interface: The function interfaces with the SpriteLoader function, and the "Open" common dialog. SpriteLoader passes the filename to this function.

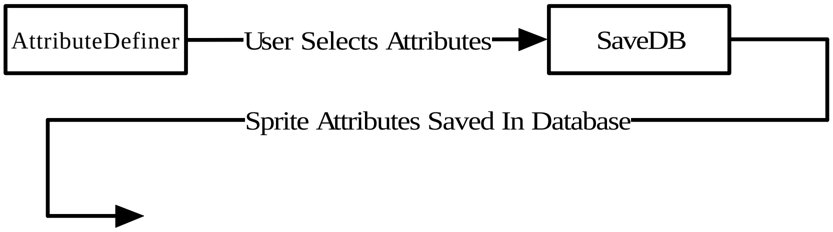
Issues: The file chosen must be a bitmap, and the SpriteLoader function must be implemented.

Constraints: Without the SpriteLoader function and the common dialog control the file will not be displayed. The common dialog control is included in the Visual Basic application wizard.

AttributeDefiner:

Narrative: Function is called after the ImageLoader is done. This is where the user is allowed to determine all of the attributes that a particular sprite will have.

Diagram:



Interface: The function interfaces with the SaveDB function. Data is saved to the Database after the user is done selecting attributes.

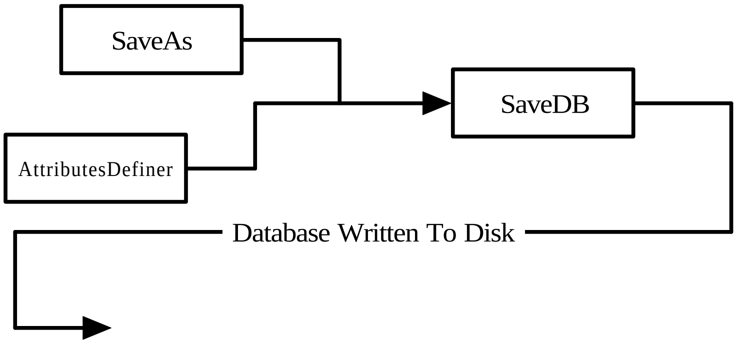
Issues: The database must be working properly.

Constraints: Without the SaveDB function the attributes will be lost.

SaveDB:

Narrative: Function is called when the user chooses Save As from the file menu, or after attributes are updated. The data is written to the Access database.

Diagram:



Interface: The function interfaces with the either the SaveAs or the AttriubuteDefiner functions. Data is saved to the Database after the user is done selecting attributes.

Issues: The database must be working properly.

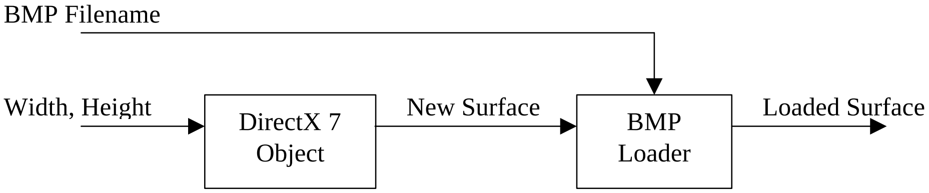
Constraints: Without the SaveDB function the changes will be lost.

DRAW_HANDLER

Create_Surface:

Narrative: Receives as input a filename, and width and a height. Based on the width and height the appropriate DirectX 7 calls are made to allocate memory for surface creation and once the memory is set, the filename is used to load the specified BMP file onto the DirectX 7 surface. The width and height are the full size of the actual image itself. Once completed creating and loading the surface, the surface is returned.

Diagram:



Interface: The function receives a filename (const char*) and a width and a height (both const int). Using these, the function interfaces with the DirectX 7 Object (DDObjectNew) to create a new DirectX 7 surface capable of holding a BMP file. Once the new surface has been created, the function opens up the specified BMP file (using filename) and makes a call to a DirectX 7 SDK defined function that loads the specified BMP onto the new surface. Once complete, the new surface is returned.

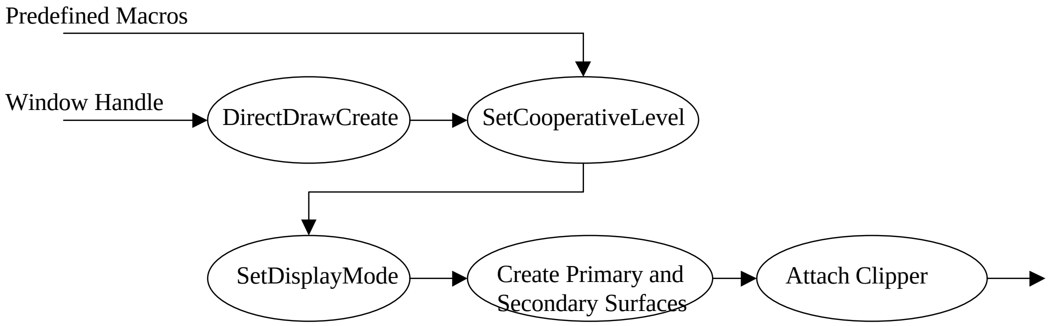
Issues: This function is not capable of determining how much video/system memory is left available. Assuming that the user loads too many images into memory, the program will most likely crash.

Constraints: The function should be able to automatically convert a different bit depth BMP to the one specified when creating the DirectX 7 object.

Ddraw_Init:

Narrative: Receives a hwnd as input (being the window handle). Using the window handle and several other predefined macros (#defines) found in a separate header file (defines.h), the DirectDraw object is first created. Once created the Cooperative level is set (how to share resources with Windows). The display mode is set (i.e. 800 X 600 X 16 bit) and the Primary surface is created. A secondary surface (back buffer) is created for use in page flipping (animation) and is attached to the primary surface. A color key is set to allow for transparent colors. Finally a clipper is attached.

Diagram:



Interface: The input to the function is a const HWND. The HWND is the handle to the window that was created in another function. Using the window handle, a DirectDraw 7 object is created. The DirectDraw 7 Object is the base object for all other drawing surfaces to be attached to. After the object is created it can be used for drawing and animation using calls defined within the DirectDraw 7 object.

Issues: If the user does not have DirectX 7 installed on his/her machine and their compiler is not linking to the correct DirectX 7 files, this function will not even compile.

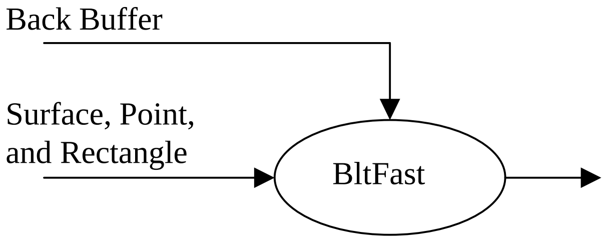
If the user tries to set the display mode to something odd, the function will crash.

Constraints: The GUI will have to limit the choices the user can make for the display mode so that no strange entries can be made that will make the system crash.

Draw_Surface:

Narrative: Receives a surface, a point and a rectangle. The surface is the surface that will be used for the graphic itself. The image has a BMP loaded onto it so that using the rectangle sent in, a portion of the BMP file can be "grabbed" at a time instead of using the entire image. The point sent in is used to determine where on the screen will the chunk of the BMP file be drawn. So essentially, a chunk of the surface is grabbed, and is repainted onto another surface at the specified coordinate.

Diagram:



Interface: A DirectDraw 7 surface is sent in as well as a const POINT and a const RECT. The DirectDraw 7 surface contains a specific BMP file already loaded onto it. Since only a portion of the BMP file is wanted, the const RECT sent in specifies what portion of the BMP to draw. The const POINT is used to reposition that portion to a specific area on the screen. In order to do so, this function has to interface with the back buffer we created in Ddraw_Init.

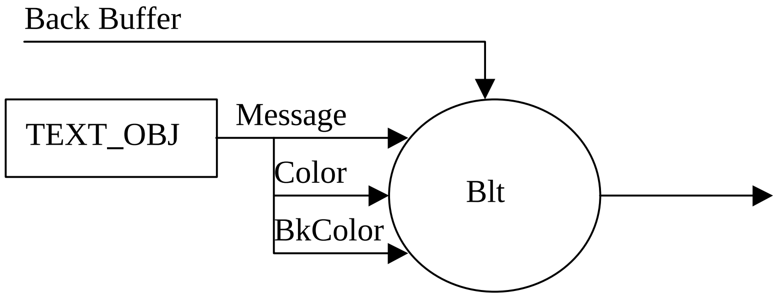
Issues: The more sprites on the screen, the more often this function will be called. Since this function is literally copying large chunks of memory from on surface to another, this function is not the fastest. So, if there are an extreme number of sprites on screen, the game itself may suffer from a performance hit.

Constraints: Due to the above issue, a clipper will have to be attached so that sprites outside of the screen will not be drawn and calls to this function will not happen nearly as often.

Draw_Text:

Narrative: Receives a Text Object as the only input. Using information within the text object, (the message itself, and the orientation on screen) the text will be output to the back-buffer surface. A background color and a foreground color are also defined within the text object, as well as whether the text is visible. This allows the user to be able to "turn off" the text.

Diagram:



Interfaces: A TEXT_OBJ is sent in as the only parameter. This function interfaces with that object in order to draw it to the back buffer. Within the object itself are the message, the background and foreground colors, the coordinate to draw it at, and whether the background is visible, or whether the text is visible at all. Once it is determined that the text is visible, the text is drawn to the back buffer (using a routine defined within the back-buffer surface).

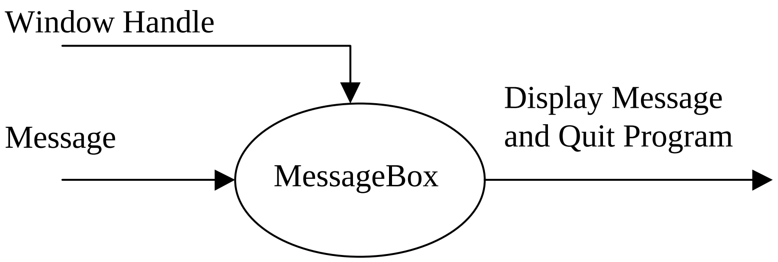
Issues: This is a fairly limited text handler because one cannot change the font that it draws. Windows chooses the current font for use when drawing the text. So only the current Windows font will be drawn.

Constraints: Text should be drawn last over every other object on the screen. This is so important information is not hidden behind an on screen sprite.

Error:

Narrative: Error is a very simple function that receives a string and outputs a text box to the screen specifying the error, and shuts down the game.

Diagram:



Interface: The only input is a const char* containing the error message to put on screen. Once on the screen the user has to click on the "OK" button or hit enter. Once clicked, the function posts a 'quit' message and the window callback function kills the application.

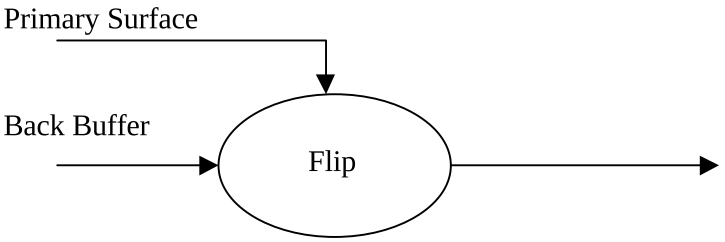
Issues: This of course causes the game to crash because it shuts down the program automatically. Another function should be made to have a similar function but not shut down the program (for minor errors).

Constraints: Possibly allow the user to attempt to continue running the program in spite of the error that happened.

Flip_Surfaces:

Narrative: Flip_Surfaces simply flips the Primary buffer and the Back-buffer. This allows for smooth animation from frame to frame. No input is sent into the function. The function first checks if the past Flip call has finished, if not it simply returns. Once this is done the surfaces will be flipped. The surfaces are then checked to make sure that their memory wasn't overwritten. If the surfaces were lost, DirectX 7 will restore the surfaces.

Diagram:



Interface: No input is sent into this function. The primary surface and the back buffer are used here. The back buffer is queried to see if a past Flip is still in progress. Once determined it either continues or cancels the function. If it continues, a call to flip the primary surface with the back buffer is made. The flip routine simply switches the pointer to the Primary surface to be swapped with that of the pointer to the back buffer. This "flips" what's being drawn in memory to the screen and vice versa.

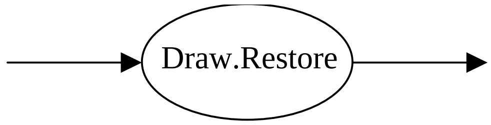
Issues: If not done correctly the surfaces could be lost, if this were to happen the screen would bomb out and nothing but the last known surface would be displayed on screen.

Constraints: To avoid losing the surfaces, once the surfaces are flipped, they should be checked to see whether they we lost. If so the surfaces need to be restore.

Restore:

Narrative: Restore receives no input. It simply checks to see if a DirectX 7 Object exists, and if so restore all surfaces. This is needed in case the surfaces get lost. This can happen when the user ALT-TABs out of the program and back in again.

Diagram:



Interface: There is no input to this function. The DirectX 7 Object is checked to see whether it exists or not. If it does exist, make a call to the restore routine (defined within the DirectX 7 object) to restore the surfaces.

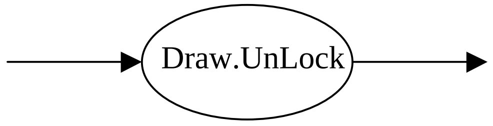
Issues: Older versions of DX made you restore each surface individually which was both time consuming for the programmer and the system itself. DirectX 7 fixes this problem with one call to its RestoreAllSurfaces function.

Constraints: It should be determined whether the DirectX 7 Object still exists. If one tied to restore surfaces that do not exist it may cause the program to crash.

Unlock:

Narrative: This function simply unlocks the Primary surface and the back buffer. This is used so that the game has the ability to be ALT-TABbed out of. By unlocking both surfaces it gives control of the video buffer back to Windows.

Diagram:



Interface: This function calls the UnLock functions defined within the primary surface and the back buffer. By doing so it frees up control of the video buffer so that Windows can write to the screen.

Issues: Unlocking the surfaces during a page flip may cause a slight animation hiccup. Most of the time the user will never see this because it will happen as they are hitting ALT-TAB.

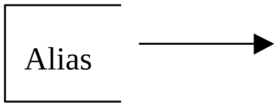
Constraints: None.

IMAGE_OBJ:

Get_Alias:

Narrative: This function simply returns the Alias of the IMAGE_OBJ. The Alias is the user given name to the surface in question.

Diagram:



Interface: This receives no input and simply sends the Alias back to wherever this function was called from.

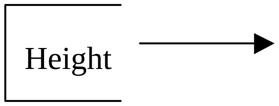
Issues: An attempt to Get_Alias an IMAGE_OBJ that has no alias set may crash the program, this should not be allowed.

Constraints: This function must check to see if the Alias for the current IMAGE_OBJ has been set before trying to send back undefined information.

Get_Height:

Narrative: This function simply returns the Height of the IMAGE_OBJ. The height is the size in pixels of the surface in question.

Diagram:



Interface: This function receives no input and simply sends back the Height to wherever this function was called from.

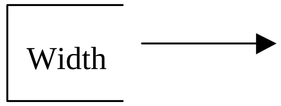
Issues: If a call is made to Get_Height and no height has been defined, the program may try to load an image to a surface that does not have enough memory allocated to contain that image. This may cause the program to crash.

Constraints: A check has to be made to make sure that the surface in question does have a Height associated with it.

Get_Width:

Narrative: This function simply returns the Width of the IMAGE_OBJ. The width is the size in pixels of the surface in question.

Diagram:



Interface: This function receives no input and simply sends back the width to wherever this function was called from.

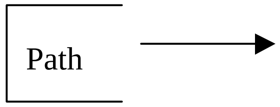
Issues: If a call is made to Get_Width and no width has been defined, the program may try to load an image to a surface that does not have enough memory allocated to contain that image. This may cause the program to crash.

Constraints: A check has to be made to make sure that the surface in question does have a width associated with it.

Get_Path:

Narrative: This function simply returns the path of the IMAGE_OBJ. The path is the filename and path of the desired BMP file to load to the surface.

Diagram:



Interface: This function receives no input and simply sends back the path the wherever this function was called from.

Issues: If no path yet exists for the current IMAGE_OBJ, an empty string will be returned. This path is used in loading the BMP to the surface. If the path is NULL, there will be no path to load to the surface and the program will crash.

Constraints: Possibly set a default image so that the program does not crash if the path has not been set. Or possibly make the image loading function not kill the program and instead continue processing.

Get_Surface:

Narrative: This function simply returns the surface pointer of the IMAGE_OBJ. The surface pointer is the pointer to the surface that will be loaded with the specified BMP file from 'Path'.

Diagram:

Interface: This function receives no input and simply returns a surface pointer to wherever this function was called from.

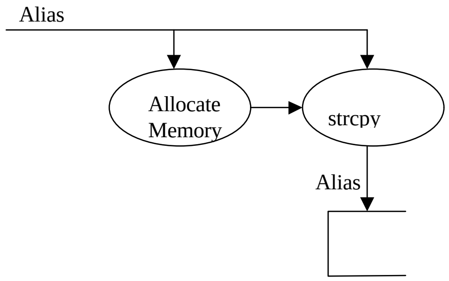
Issues: The sprites in the game rely upon these surfaces for use in drawing. If the surface was not loaded correctly the game output will either not work at all or not work correctly.

Constraints: None.

Set_Alias:

Narrative: This function receives as input a string. The string is the user-defined name that will represent this surface. This is so each sprite defined can be pointed to whichever surface needed via their alias.

Diagram:



Interface: This function receive as input a const char* containing the user-defined alias to the current surface. First memory is allocated to contain the string, and then the string is copied in using strcpy (found in string.h).

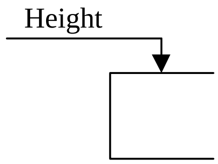
Issues: Resetting an alias may cause problems when using strcpy. The alias may end up being not what was originally sent in.

Constraints: The alias should be kept short, but this will not be enforced, as it will be simpler to just let it be the size it wants to be. Since the alias is user-defined and used within the data files, most users will see that by creating a long alias to begin with just makes them have to type more later on.

Set_Height:

Narrative: This function receives as in put an integer. The integer is the height of the BMP file to be loaded onto the surface. It will be used to correctly specify the size of the surface so that none of the image is lost.

Diagram:



Interface: This function receives a const int as its input. The integer is simply set to the current IMAGE_OBJ's Height member.

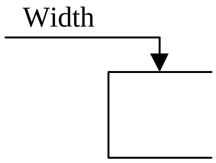
Issues: This will only accept integers as a value because the integer represents the number of pixels tall the image to be loaded will be.

Constraints: None.

Set_Width:

Narrative: This function receives as in put an integer. The integer is the width of the BMP file to be loaded onto the surface. It will be used to correctly specify the size of the surface so that none of the image is lost.

Diagram:



Interface: This function receives a const int as its input. The integer is simply set to the current IMAGE_OBJ's Width member.

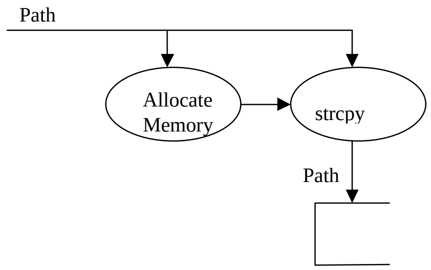
Issues: This will only accept integers as a value because the integer represents the number of pixels wide the image to be loaded will be.

Constraints: None.

Set_Path:

Narrative: This function receives as input a string. The string the filename and path of the filename of the BMP file to be loaded onto the surface. The string is set to the IMAGE_OBJ Path member.

Diagram:



Interface: This function receives a const char* as its input. First enough memory is allocated to contain the string. Then the string is simply set to the current IMAGE_OBJ's Path member. This will be used later to load the surface with the specified filename.

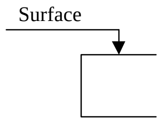
Issues: Resetting a path may cause problems when using strcpy. The path may end up being not what was originally sent in. This may in turn cause problems when loading the BMP file to the surface.

Constraints: The path should contain a .BMP extension. If the path specified does not contain .BMP, then the path is invalid and an error should be output.

Set_Surface:

Narrative: This function receives as input a pointer to a DirectX 7 surface. This surface will be used to grab portions of when animating sprites within the game. The IMAGE_OBJ member Surface is simply set to point to the temporary surface sent in.

Diagram:



Interface: This function receives a const LPDIRECTDRAWSURFACE7 as its input type. The Surface member within the current IMAGE_OBJ is simply set to point to the same memory location as that of the surface sent in.

Issues: Users that do not have DirectX 7 installed on their machine will not be able to compile nor run this particular function.

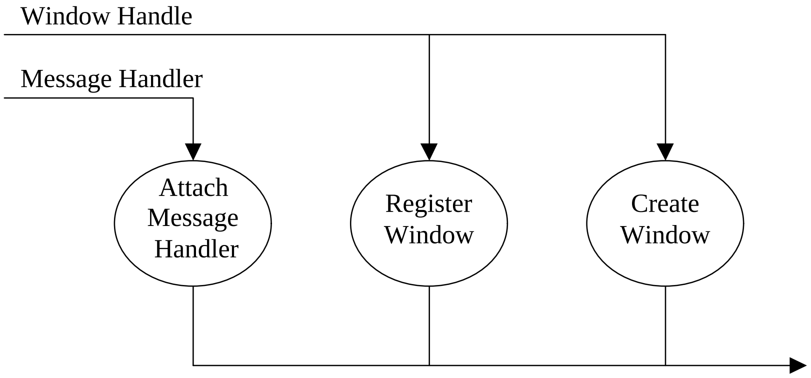
Constraints: None.

OBJ_HANDLER:

Create_Window:

Narrative: This function receives as input the name of the event handler, and the handle to the instance of the game application. Using these inputs and several predefine macros (found in defines.h) the actual window for the game is created. First several things are defined such as background color and application icon. Once all is set, the window is registered. After the window is registered the window is then created. Once created the cursor is hidden and the window is set into focus.

Diagram:



Interface: The functions inputs are two void* types. One is the handle to the event handler, and the other is the handle to the application instance. Once the event handler has been attached to the window being created, the window is registered. This attempts to register the window with Windows itself. If successful the application window is then created. If the creation of the window is successful, the window is brought to focus and the function exits. If the registering or creation of the window fail, the program will not run.

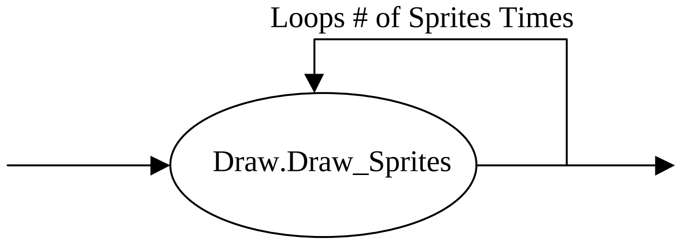
Issues: The predefined window that will be created will be a full screen popup window. The window will have no close or minimize buttons so there must be a keystroke that will end the application.

Constraints: This is the function that defines the window width and height and therefore must take into account different screen resolutions. This can be solved by using predefine macros and simply having the GUI of the design application make changes to the macros.

Draw_Sprites:

Narrative: This function receives no input. Its function is to draw all of the sprites currently created on the screen in the position that they are supposed to be drawn at. It contains a loop that continues until all sprites have been drawn.

Diagram:



Interface: This function has no input. It uses data members that are already defined within the OBJ_HANDLER. The function contains a for loop that continues to loop until its reached the number of sprites that are defined. Within the loop are calls to the current sprite. The calls get information about what part of the image to draw, and where to draw it on screen. Once that information is gathered, a call to *Draw.Draw_Surface* is called which actually draws the surface by blitting to the back buffer.

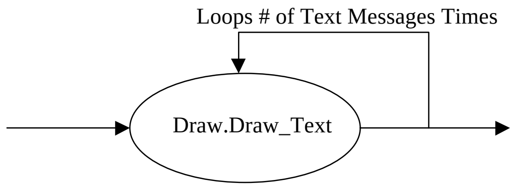
Issues: The more sprites that exist on screen, the more often the function *Draw.Draw_Surface* has to be called. *Draw.Draw_Surface* when called too often with surfaces that are large, WILL eventually cause the games graphics to stutter.

Constraints: Limit the total number of sprite to a suitable number so that the call to *Draw.Draw_Surface* is minimized.

Draw_Text:

Narrative: This function receives no input. Its function is to draw all text objects on the screen in the position that they are supposed to be drawn at. It contains a loop then continues until all text objects are drawn.

Diagram:



Interface: This function receives no input. This is a fairly simple function in that all it does is loop until all TEXT_OBJ's have been accounted for. Within the loop is a call to *Draw.Draw_Text* (which actually does the drawing).

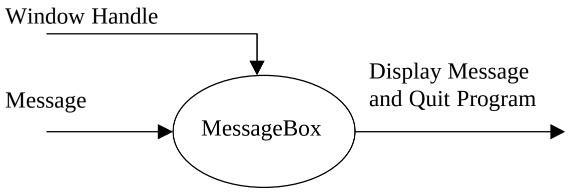
Issues: If all TEXT_OBJ are define as invisible, this will loop through each object anyway. This of course eats up some processor time and will produce no output.

Constraints: None.

Error:

Narrative: Error is a very simple function that receives a string and outputs a text box to the screen specifying the error, and shuts down the game.

Diagram:



Interface: The only input is a const char* containing the error message to put on screen. Once on the screen the user has to click on the “OK” button or hit enter. Once clicked, the function posts a 'quit' message and the window callback function kills the application.

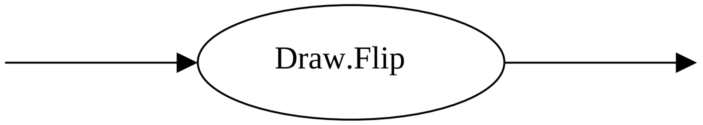
Issues: This of course causes the game to crash because it shuts down the program automatically. Another function should be made to have a similar function but not shut down the program (for minor errors).

Constraints: Possibly allow the user to attempt to continue running the program in spite of the error that happened.

Flip:

Narrative: Flip receives no input. Its function is simply a pipeline to the *Draw.Flip* function that actually does the page flipping (animation).

Diagram:



Interface: This function is a way to be able to flip the surfaces from the main program without directly interacting with the *Draw* object. The *Draw* object is contained within the OBJ_HANDLER and so therefore is a private member. The only way to use *Draw*’s internal function is to have a function in the OBJ_HANDLER to call the specified *Draw* function.

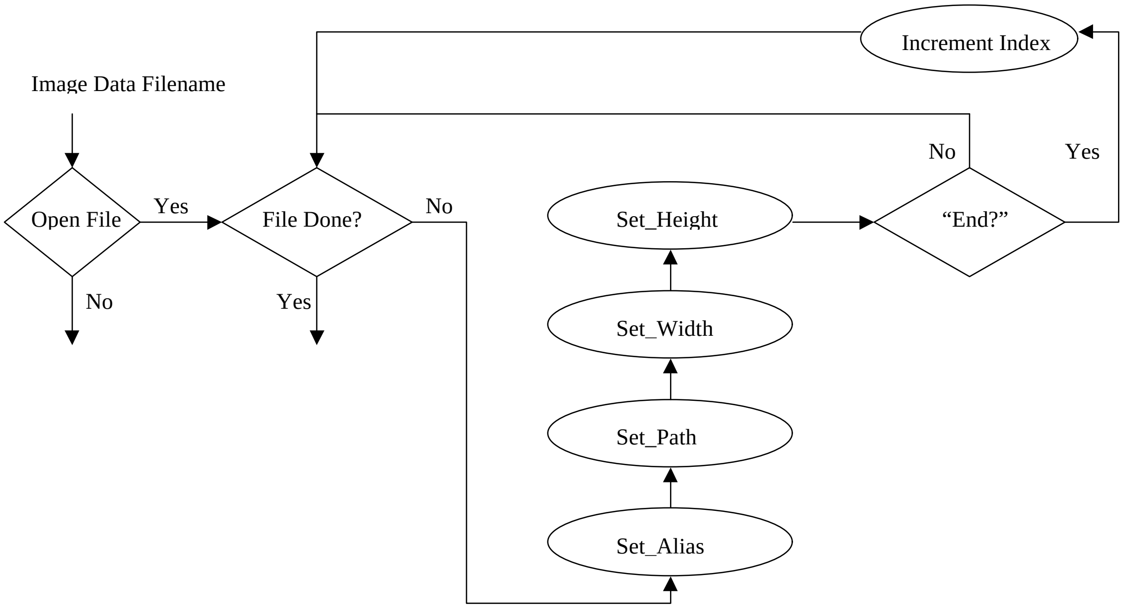
Issues: None.

Constraints: None.

Load_Image:

Narrative: This function receives a string as input. That string is the handle to a filename that is a data file containing information about every IMAGE_OBJ that will be within the game. The filename given will be opened. If the file does not exist, the function will open a file for output and will output an error message to a log file. If the file opening was successful, the function will continue to loop until all data from the file has been read in. Within the loop the file is parsed and each piece of information is used to load an IMAGE_OBJ with all of the correct information it needs. When the file is empty, the file is closed.

Diagram:



Interface: This function receives a const char* as input. The string is the handle to a filename that contains all of the information needed to fill an IMAGE_OBJ object. An attempt to open that file is made. If the file exists in the specified path, the program will continue. If not, an output file is opened and an error message is output. Once a valid file is opened, each line of data is read in one by one. Each line will consist of an operation and an operator. The operand is parsed off and the operator tells what function to use to set the data with. Once an "end" operation has been read, the image counter is incremented and a new IMAGE_OBJ is created. After all IMAGE_OBJ are read in from the file, the loop ends, and the file is closed.

Issues: If the data file's contents are not set up correctly by the GUI builder or if the user edits the data file incorrectly, some or all of the information in the file will not be loaded correctly. This could end up in undefined IMAGE_OBJ's and cause problems in loading sprites correctly.

If the user decided to move the BMP files the a different path or filename, the data file would have no way of knowing this and would either have to be hand edited by the user, or would have to be re-edited by the GUI builder.

If the user wants to hand edit the data file, they will have to be aware of the correct syntax expected by this function. Complete syntax knowledge with examples should be easily accessible in the help files.

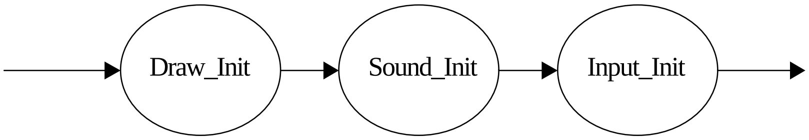
The string handling in C++ is something left to be desired. With a more flexible language the data files could be more fault tolerant than they will be, but with the limited string handling and file streaming of C++, the data files will have to be very correct.

Constraints: The file handling part will have to be somewhat flexible to allow for added white space. This is essential. All users have different ideas of how separate items should be spaced and therefore this function will have to take that into account.

Init_All:

Narrative: This function accepts no inputs. Its duty is to simply call the functions that will in turn create the DD, DS, and DI objects.

Diagram:



Interface: This function makes a call *to Draw.DDraw_Init, Sound.DSound_Init*, and *Input.DInput_Init.* Each of these functions are DirectX 7 functions that will create the draw, sound, and input objects to be used in the game.

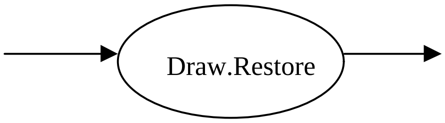
Issues: If the user does not have DirectX 7 installed on their system, the functions being called will not compile nor run correctly.

Constraints: None.

Restore:

Narrative: Restore receives no input. Its function is simply a pipeline to the *Draw.Restore* function that actually restores lost surfaces.

Diagram:



Interface: This function is a way to be able to restore lost surfaces from the main program without directly interacting with the *Draw* object. The *Draw* object is contained within the OBJ_HANDLER and so therefore is a private member. The only way to use *Draw*'s internal function is to have a function in the OBJ_HANDLER to call the specified *Draw* function.

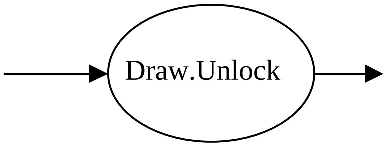
Issues: None.

Constraints: None.

Unlock:

Narrative: Unlock receives no input. Its function is simply a pipeline to the *Draw.Unlock* function that actually unlocks the primary surface and the back buffer.

Diagram:



Interface: This function is a way to be able to unlock the primary surface and the back-buffer main program without directly interacting with the *Draw* object. The *Draw* object is contained within the OBJ_HANDLER and so therefore is a private member. The only way to use *Draw*'s internal function is to have a function in the OBJ_HANDLER to call the specified *Draw* function.

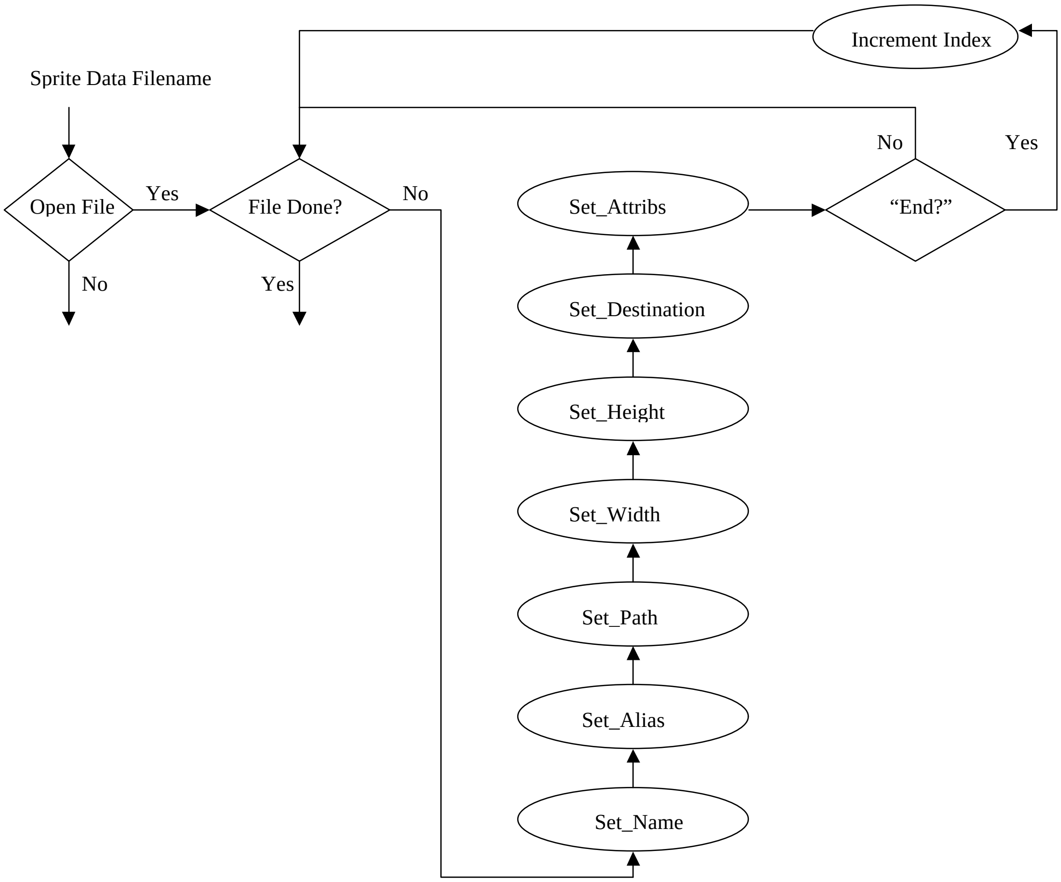
Issues: None.

Constraints: None.

Load_Sprite:

Narrative: This function receives a string as input. That string is the handle to a filename that is a data file containing information about every SPRITE_OBJ that will be within the game. The filename given will be opened. If the file does not exist, the function will open a file for output and will output an error message to a log file. If the file opening was successful, the function will continue to loop until all data from the file has been read in. Within the loop the file is parsed and each piece of information is used to load an SPRITE_OBJ with all of the correct information it needs. When the file is empty, the file is closed.

Diagram:



Interface: This function receives a const char* as input. The string is the handle to a filename that contains all of the information needed to fill a SPRITE_OBJ object. An attempt to open that file is made. If the file exists in the specified path, the program will continue. If not, an output file is opened and an error message is output. Once a valid file is opened, each line of data is read in one by one. Each line will consist of an operation and an operator. The operand is parsed off and the operator tells what function to use to set the data with. Once an "end" operation has been read, the image counter is incremented and a new SPRITE_OBJ is created. After all SPRITE_OBJ are read in from the file, the loop ends, and the file is closed.

Issues: If the data file's contents are not set up correctly by the GUI builder or if the user edits the data file incorrectly, some or all of the information in the file will not be loaded correctly. This could end up in undefined SPRITE_OBJ's and cause problems in loading sprites correctly or at all.

If the user decided to change the BMP files in any way, (i.e. resizing or changing position of entities within), the data file would have no way of knowing this. This means that the user would have to hand edit the data file to compensate for the changes, or the data file would have to be re-edited by the GUI builder.

If the user wants to hand edit the data file, they will have to be aware of the correct syntax expected by this function. Complete syntax knowledge with examples should be easily accessible in the help files.

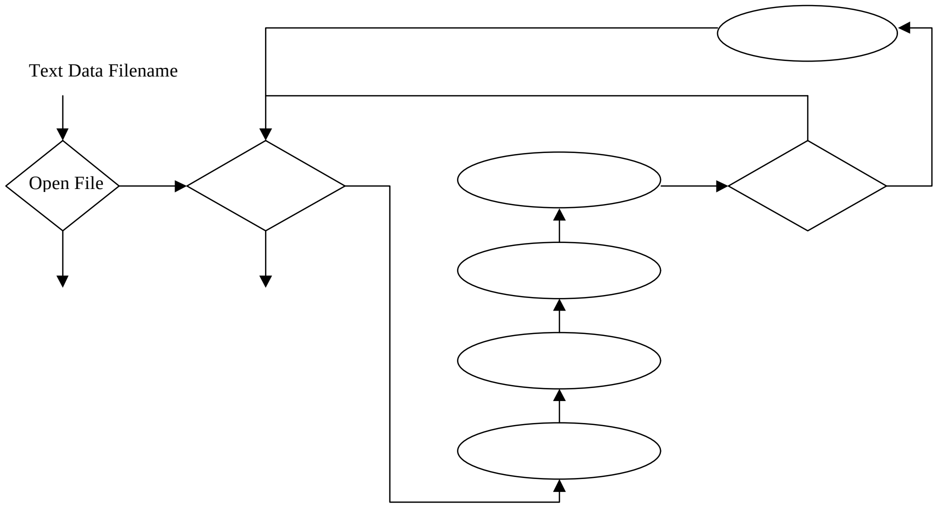
The string handling in C++ is something left to be desired. With a more flexible language the data files could be more fault tolerant than they will be, but with the limited string handling and file streaming of C++, the data files will have to be very correct.

Constraints: The file handling part will have to be somewhat flexible to allow for added white space. This is essential. All users have different ideas of how separate items should be spaced and therefore this function will have to take that into account.

Load_Text:

Narrative: This function receives a string as input. That string is the handle to a filename that is a data file containing information about every TEXT_OBJ that will be within the game. The filename given will be opened. If the file does not exist, the function will open a file for output and will output an error message to a log file. If the file opening was successful, the function will continue to loop until all data from the file has been read in. Within the loop the file is parsed and each piece of information is used to load an TEXT_OBJ with all of the correct information it needs. When the file is empty, the file is closed.

Diagram:



Interface: This function receives a const char* as input. The string is the handle to a filename that contains all of the information needed to fill an TEXT_OBJ object. An attempt to open that file is made. If the file exists in the specified path, the program will continue. If not, an output file is opened and an error message is output. Once a valid file is opened, each line of data is read in one by one. Each line will consist of an operation and an operator. The operand is parsed off and the operator tells what function to use to set the data with. Once an "end" operation has been read, the image counter is incremented and a new TEXT_OBJ is created. After all TEXT_OBJ are read in from the file, the loop ends, and the file is closed.

Issues: If the data file's contents are not set up correctly by the GUI builder or if the user edits the data file incorrectly, some or all of the information in the file will not be loaded correctly. This could end up in undefined TEXT_OBJ's and cause problems in loading the text sprites correctly or at all.

If the user wants to hand edit the data file, they will have to be aware of the correct syntax expected by this function. Complete syntax knowledge with examples should be easily accessible in the help files.

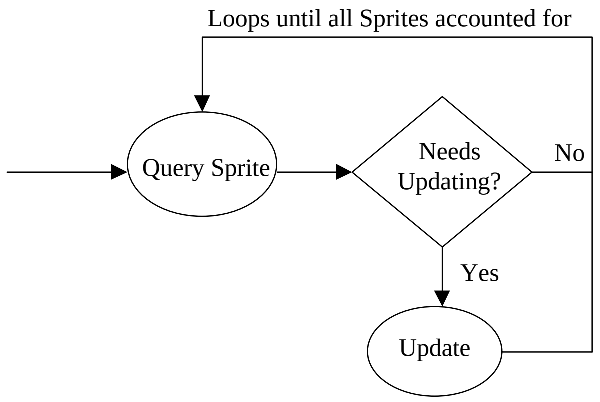
The string handling in C++ is something left to be desired. With a more flexible language the data files could be more fault tolerant than they will be, but with the limited string handling and file streaming of C++, the data files will have to be very correct.

Constraints: The file handling part will have to be somewhat flexible to allow for added white space. This is essential. All users have different ideas of how separate items should be spaced and therefore this function will have to take that into account.

Update_Frame:

Narrative: This function accepts no inputs. Its duty is to update each sprite's (that has more than one frame of animation) frame of animation. This will be done either by using a millisecond timer, or by setting a frame rate for each particular sprite. This function contains a for loop that will query each sprite and determine whether their frame needs updating.

Diagram:



Interface: This function has no inputs. It contains a for loop that loops until all sprites have been accounted for. During each iteration, the current sprite is queried to see if their frame of animation needs updating. If it does need updating (checked by time passed or frame count), the source coordinate for the sprite are changed so that when the sprite is drawn the image will be different than it was on the previous frame.

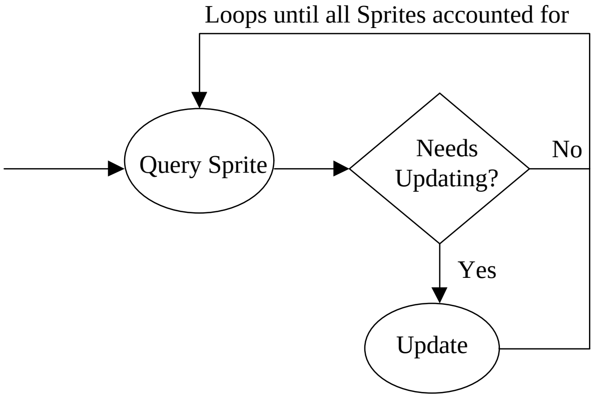
Issues: While it's not necessary to update the frame information for sprites off screen, this will be done anyway. The reason is simply because it does not take that much processing power to do so and it's easier to do all of the sprites instead of keeping track of which ones are on screen and which are off.

Constraints: Some objects animation patterns will vary greatly as the game moves on. Other objects animation patterns will have to be based on user input and not by (or in addition to) timer or frame count.

Update_Dest:

Narrative: This function receives no inputs. Its duty is to update the screen position of every sprite that needs repositioning. Querying each sprite to see how each one moves will do this. Some will be user controlled, others will have their own simple AI routines, and others will simply not move.

Diagram:



Interface: This function has no inputs. It contains a for loop that loops until each sprite has been accounted for. During each iteration, the sprite is queried to determine how to update the screen destination, if at all. Each sprite is updated to its new destination by a certain "velocity" also contained within the sprite.

Issues: While it's not necessary to update the screen destination for sprites off screen, this will be done anyway. The reason is simply because it does not take that much processing power to do so and it's easier to do all of the sprites instead of keeping track of which ones are on screen and which are off.

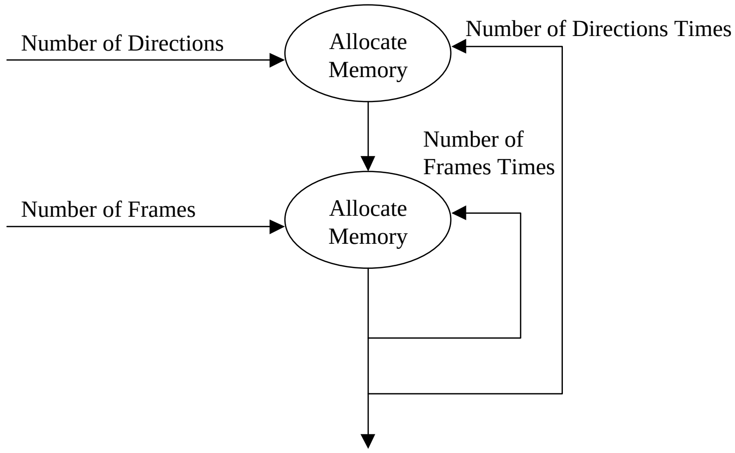
Constraints: Some objects motion patterns will vary different from others. This will have to be taken into account when writing this function. This may have to call separate AI routines to update each sprite's position accordingly.

SPRITE_OBJ:

Init_Source:

Narrative: This function receives the number of directions and number of frames of animation for the spite being defined. These allow the function to allocated enough memory to contain the matrix of coordinates that will be defined in a later function. For example, if the sprite being defined has only 1 static image then this function will allocate enough memory for only 1 coordinate. This function does not actually set the coordinates, it simply creates enough room to store them later on.

Diagram:



Interface: This function receives two const int as input. These integers represent the number of directions the current sprite will have (i.e. 1,2,4,8) and the number of frames of animation per direction (i.e. 1,2,3,4…). If the input values have a value of 4 for directions and 3 for frames, that means this sprite has 4 directions (N, W, E, S) and per each direction there are 3 animations associated with it. This function has two loops inside. The outer loop will loop as until all directions have been accounted for, and the inner loop will loop until all frames per direction are accounted for. During each iteration, new memory is created as a placeholder for coordinates that will be defined in a later function.

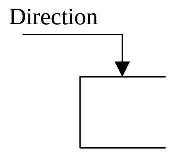
Issues: The total number of frames (directions * frames) will have to be known at this time. If the user decides to add another frame/direction to the data file, they will also have to change the number of directions or the number of frames. This should be outlined in the help file.

Constraints: This will have to be flexible enough to allow for vast differences in numbers. A user may want to have 64 animations per direction and so the array cannot simply be a static array, it will have to be dynamically allocated to allow for this.

Update_Dir:

Narrative: This function receives a direction to set the sprite to. The direction is simply an integer used to look op the correct row in the matrix or coordinates that was defined earlier. This function will be called any time the sprite changes directions.

Diagram:



Interface: This function receives one const int for input. This integer is the index of a particular row in the matrix of coordinates that has been predefined. This will simply set the member Current_Dir to be whatever direction the user sends in.

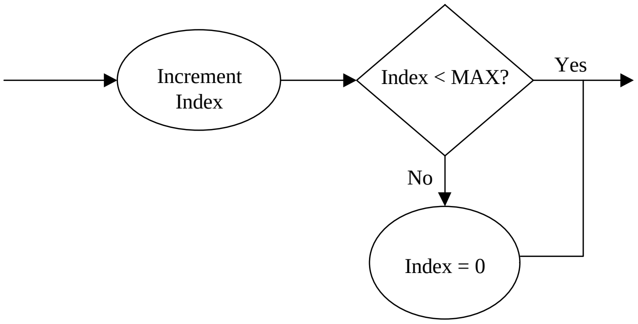
Issues: An invalid direction could be sent in. If this happens the program will try to access memory that does not exist. This will either cause the game to not draw a sprite on the screen, or could cause the system to crash.

Constraints: There should be a test within this function to ensure that the direction sent in is indeed a valid direction. If not, simply keep the current direction. If so, update the current direction to the specified direction.

Update_Frame:

Narrative: This function receives no inputs. This simply increments the index of the column in the coordinate matrix to point to the next frame of animation. Once the last frame has been reached, the index is reset to zero.

Diagram:



Interface: This function receives no inputs. Internally, the member Current_Frame is incremented to the next frame. If after incrementing the index exceeds the number of frames defined, the frame will be reset to zero to begin the animation again.

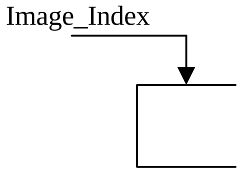
Issues: This does not allow for different patterns of animation. Each animation will load in succession as they were defined in the data file. If the user wants to repeat a certain frame, they will have to either edit the data file and add the same coordinates twice, or use the GUI builder the choose that repeated frame.

Constraints: Make sure that the function is not attempting to draw a frame that does not exist. This may simply cause a hiccup in animation, or could possibly crash the system.

Set_Image_Index:

Narrative: This function receives an index to the IMAGE_OBJ array. This is so that the sprite actually has a portion of a BMP file associated with it. Without this image index, no graphics would be drawn on screen. The index sent in is simply set to be the member Image_Index.

Diagram:



Interface: This function receives a const int as input. The integer is an index to the IMAGE_OBJ array. The index points to a certain surface so that at run-time, a graphic will be associated with the sprite.

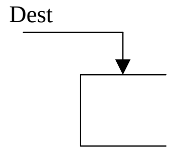
Issues: If the index sent in is an invalid index, the game will attempt to blit from and image that is not defined. This will either cause the sprite to not be drawn at all, or will cause the system to crash.

Constraints: The image index will have to be pre-processed to determine whether it is indeed a valid index.

Set_Dest:

Narrative: This function receives a destination coordinate as input. The coordinate is the pixel on the screen to draw the sprite at. The coordinate sent in is simply set to the member Dest_Coord.

Diagram:



Interface: This function receives a const POINT as input. The POINT is actually a pixel coordinate of where to draw the object on screen. The coordinate sent in is simply set to the member Dest_Coord.

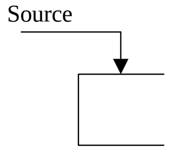
Issues: Since sprites can be positioned off screen, the user may mistakenly put the wrong coordinates for an object. This means a sprite that they expect to see on screen will not be at all. This is unavoidable. The user will have to move the coordinates of the sprite in the GUI builder or simply change the data file.

Constraints: Some sprites should be able to have coordinates off screen. This will allow for objects to move onto the screen at a later time. This means that a clipper will have to be attached so that unnecessary drawing if off screen sprites is not done. This also means pixel values that do not fit within the current screen resolution ARE valid entries.

Set_Source:

Narrative: This function receives a coordinate as input. The coordinate is used to determine what pixel on the specified BMP file to start blitting from. Some sprites will have more than one frame of animation and therefore this function will have to set the correct index in the array of coordinates.

Diagram:



Interface: This function receives a const POINT as input. The POINT is a pixel position on the BMP file that will be used during the blit operation to grab a certain portion of the BMP file. This function will set the current index of the source coordinate array to be the coordinate sent in, and will also increment the index of the array so that the next call (if there is a next call) will place the new coordinate in the correct index.

Issues: Init_Source allocates enough memory to hold a certain range of source coordinates. The number of directions and the number of frames determine this. If the user decides to input an additional source coordinate (frame of animation) without changing the number of frames member, the system will attempt to load the coordinate, but will fail.

Constraints: None.

Set_Name:

Narrative: This function receives a name as input. The name is used to distinguish between sprites. This is primary used in the GUI builder but could be used in the game itself as well. The name is copied into the member Name.

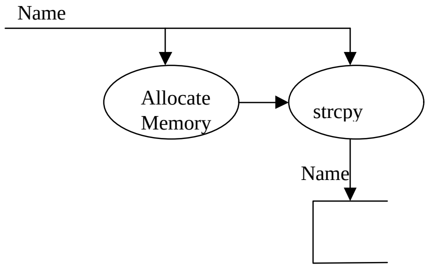


Diagram:

Interface: This function receives a const char* as input. The function first allocates memory the size of the string sent in. Once allocated, the function uses strcpy (string.h) to copy the string into the data member Name.

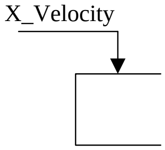
Issues: None.

Constraints: None.

Set_X_Velocity:

Narrative: This function receives a velocity as input. The velocity is simple the number of pixels to the left or right that the sprite will move. This should default to zero. The velocity sent in will simply be set to the data member X_Velocity.

Diagram:



Interface: This function receives a const int as input. The integer represents the number of pixels to offset the current SPRITE_OBJ by on the screen (or off the screen). The integer is simply set to the data member X_Velocity.

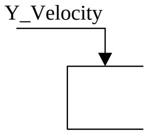
Issues: If the velocity sent in is very large, the sprite itself may not appear after its first frame. This would be because the sprite shot off the screen at a very large rate that it would only appear onscreen for the first few frames.

Constraints: Negative and positive velocities should be allowed. Negative will move the sprite to the left, positive to the right.

Set_Y_Velocity:

Narrative: This function receives a velocity as input. The velocity is simple the number of pixels up or down that the sprite will move. This should default to zero. The velocity sent in will simply be set to the data member Y_Velocity.

Diagram:



Interface: This function receives a const int as input. The integer represents the number of pixels to offset the current SPRITE_OBJ by on the screen (or off the screen). The integer is simply set to the data member Y_Velocity.

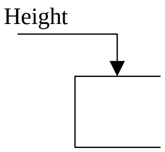
Issues: If the velocity sent in is very large, the sprite itself may not appear after its first frame. This would be because the sprite shot off the screen at a very large rate that it would only appear onscreen for the first few frames.

Constraints: Negative and positive velocities should be allowed. Negative will move the sprite up, positive down.

Set_Height:

Narrative: This function receives as in put an integer. The integer is the height of the sprite itself. It will be used so that the sprite will remain at its correct size and proportion when blitting to the screen. This will also be used to determine how much of the BMP file to blit.

Diagram:



Interface: This function receives a const int as its input. The integer is simply set to the current SPRITE_OBJ's Height member.

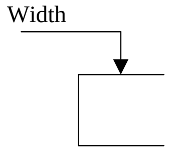
Issues: This will only accept integers as a value because the integer represents the number of pixels tall the image to be loaded will be.

Constraints: None.

Set_Width:

Narrative: This function receives as in put an integer. The integer is the height of the sprite itself. It will be used so that the sprite will remain at its correct size and proportion when blitting to the screen. This will also be used to determine how much of the BMP file to blit

Diagram:



Interface: This function receives a const int as its input. The integer is simply set to the current SPRITE_OBJ's Width member.

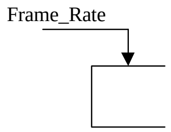
Issues: This will only accept integers as a value because the integer represents the number of pixels wide the image to be loaded will be.

Constraints: None.

Set_Frame_Rate:

Narrative: This function receives a frame rate as its only input. The frame rate is the 'timer' that will be used to determine when a sprite's frame of animation needs to be updated. Without this, the sprite's frames would rapidly flip through too fast for the user to see.

Diagram:



Interface: This function receives a const int as its input. The integer is the number of frames to wait until updating the sprite's source coordinates. The integer is simply set to the data member Frame_Rate.

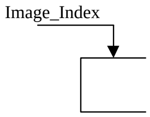
Issues: This method of controlling animation will be directly coupled to the computer that the game is running on. The better the computer, the higher the frame rate of the game, the higher the frame rate of the game, the faster the animation will seem. We might also include a true millisecond timer based animation cycle.

Constraints: Negative numbers could possible be used to be a 'stop' for the animation itself. This may prove useful for user controlled sprites.

Get_Image_Index:

Narrative: This function receives no input. Its duty is to simply return the SPRITE_OBJ's image index to wherever the function was called from. The image index will be used when blitting the sprite to the screen.

Diagram:



Interface: The function does nothing more than return the data member Image_Index's value to wherever it was called form.

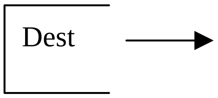
Issues: If the image index is undefined, this function will still return a value. The value returned may not correspond to an image at all and may cause the game to crash upon attempting to blit this sprite.

Constraints: This should definitely have a default value of zero. That way more that likely the index will point to SOME BMP file (even though it may not be the desired one). This will help to eliminate some blitting problems.

Get_Dest:

Narrative: This function receives no input. Its duty is to simply return the current SPRITE_OBJ's destination coordinate (drawing coordinate) to wherever it was called from.

Diagram:



Interface: This function simply returns the data member Dest_Coord to wherever it was called from. This coordinate is used to blit the sprite to the screen.

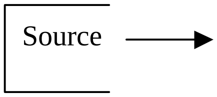
Issues: If the destination coordinate is not defined, the sprite may be drawn in odd coordinates on the screen (or may not be drawn at all).

Constraints: The data member Dest_Coord should have a default value of (0, 0). This will ensure that all objects will be drawn (although not at the right place if not specified).

Get_Source:

Narrative: This function receives no input. Its duty is to simply return the current SPRITE_OBJ's current source coordinate (where to blit from) to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member Source_Coord at the current index (determined by Current_Dir and Current_Frame). This coordinate is used to determine what frame to blit to the screen.

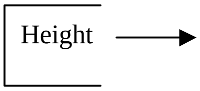
Issues: If Current_Dir and Current_Frame are undefined, the desired output will not be displayed and may cause the game to crash. Care has to be taken that Set_Source is called prior to any call to this function.

Constraints: None.

Get_Height:

Narrative: This function simply returns the Height of the SPRITE_OBJ. The height is the size in pixels of the sprite in question.

Diagram:



Interface: This function receives no input and simply sends back the Height to wherever this function was called from.

Issues: If a call is made to Get_Height and no height has been defined, the program will attempt to draw regardless. This will not be a problem unless somehow the Height was by default a negative number, which could have undesired affects. This may cause the program to crash.

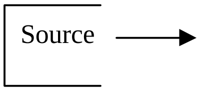
If the height specified exceeds the height of the BMP file itself, the game will most likely produce undesired output.

Constraints: A check has to be made to make sure that the surface in question does have a width associated with it and that the width be a positive integer, and smaller than the image itself.

Get_Width:

Narrative: This function simply returns the Width of the SPRITE_OBJ. The width is the size in pixels of the surface in question.

Diagram:



Interface: This function receives no input and simply sends back the width to wherever this function was called from.

Issues: If a call is made to Get_Width and no height has been defined, the program will attempt to draw regardless. This will not be a problem unless somehow the width was by default a negative number, which could have undesired affects. This may cause the program to crash.

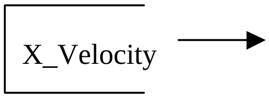
If the width specified exceeds the width of the BMP file itself, the game will most likely produce undesired output.

Constraints: A check has to be made to make sure that the surface in question does have a width associated with it and that the width be a positive integer, and smaller than the image itself.

Get_X_Velocity:

Narrative: This function receives no inputs. Its duty is to simply return the data member X_Velocity to wherever this function was called from.

Diagram:



Interface: This function receives no inputs and simply sends back the current SPRITE_OBJ's x velocity to wherever this function was called from.

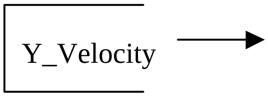
Issues: The velocity could be set to a very large number. If this happened and the number was returned, it would be used to update the current SPRITE_OBJ's x position on the screen. This may in fact move the sprite so much that it would no longer be on the screen.

Constraints: None.

Get_Y_Velocity:

Narrative: This function receives no inputs. Its duty is to simply return the data member Y_Velocity to wherever this function was called from.

Diagram:



Interface: This function receives no inputs and simply sends back the current SPRITE_OBJ's y velocity to wherever this function was called from.

Issues: The velocity could be set to a very large number. If this happened and the number was returned, it would be used to update the current SPRITE_OBJ's y position on the screen. This may in fact move the sprite so much that it would no longer be on the screen.

Constraints: None.

Get_Frame_Rate:

Narrative: This function receives no input. Its duty is to simply return the data member Frame_Rate to wherever this function was called from.

Diagram:

Interface: This function receives no input and simply returns the current SPRITE_OBJ's frame rate to wherever this function was called from.

Issues: If the frame rate is set too high the sprite will appear to have no animation, if too low, it will animate too fast.

Constraints: None.

Get_Rate_Count:

Narrative: This function receives no input. Its duty is to simply return the data member Rate_Count to wherever this function was called from.

Diagram:

Interface: This function receives no input and simply returns the current SPRITE_OBJ's rate count to wherever this function was called from.

Issues: None.

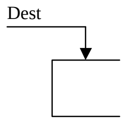
Constraints: The rate count should fall somewhere between zero and the frame rate. A check should be made to make sure that the value does indeed fall within this range.

TEXT_OBJ:

Set_Dest:

Narrative: This function receives a destination coordinate as input. The coordinate is the pixel on the screen to draw the TEXT_OBJ at. The coordinate sent in is simply set to the data member Dest_Coord.

Diagram:



Interface: This function receives a const POINT as input. The POINT is actually a pixel coordinate of where to draw the TEXT_OBJ on screen. The coordinate sent in is simply set to the data member Dest_Coord.

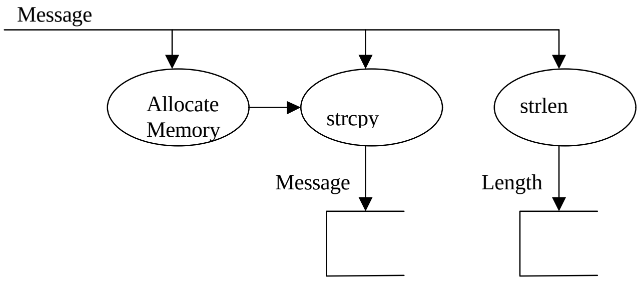
Issues: Since TEXT_OBJ's can be positioned off screen, the user may mistakenly put the wrong coordinates for an object. This means text that they expect to see on screen will not be at all. This is unavoidable. The user will have to move the coordinates of the text in the GUI builder or simply change the data file.

Constraints: Some TEXT_OBJ's should be able to have coordinates off screen. This will allow for objects to move onto the screen at a later time. This means that a clipper will have to be attached so that unnecessary drawing if off screen TEXT_OBJ's is not done. This also means pixel values that do not fit within the current screen resolution ARE valid entries.

Set_Message:

Narrative: This function receives a message as input. The message is an actual text message the will be output to the screen. The message is copied into the data member Message. Once the message is set, the length of the message is determined and set to the data member Length.

Diagram:



Interface: This function receives a const char* as input. The function first allocates memory the size of the string sent in. Once allocated, the function uses strcpy (string.h) to copy the string into the data member Message. Once the message is set, strlen (string.h) is called to determine the length of the message. The length is then set to the data member Length.

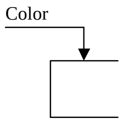
Issues: None.

Constraints: When reading in the message from the data file, the file handler has to be able to accept spaces in the message. This should be done using the getline function (fstream.h).

Set_Color:

Narrative: This function receives a color as input. The color is used to specify what color the text message will be drawn on the screen with. The color is copied into the data member Color.

Diagram:



Interface: This function receives a const COLORREF as input. The COLORREF is a Windows data type that holds an RBG color value. The color is set to the data member Color.

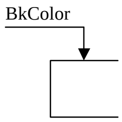
Issues: If the game is running at a color depth other than 16 bit, Windows will automatically attempt to adjust the color for the correct color depth.

Constraints: None.

Set_BkColor:

Narrative: This function receives a color as input. The color is used to specify with what color the text message will have as its background color. The color is copied into the data member BkColor.

Diagram:



Interface: This function receives a const COLORREF as input. The COLORREF is a Windows data type that holds an RBG color value. The color is set to the data member BkColor.

Issues: If the game is running at a color depth other than 16 bit, Windows will automatically attempt to adjust the color for the correct color depth.

Constraints: None.

Set_Trans:

Narrative: This function receives a flag as its input. The flag is used to specify whether or not the background of the text message is transparent or opaque. The flag is copied as a Boolean value into the data member Transparent.

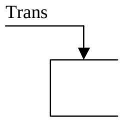


Diagram:

Interface: This function receives a const int as its input. The integer can be set at one or zero, which determines if the background of the current TEXT_OBJ is transparent or opaque. The integer is copied into the data member Transparent.

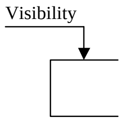
Issues: The background color (BkColor) is unnecessarily set if this flag is set to transparent. The background color will never be seen unless this flag is reset to opaque.

Constraints: Values other than one or zero should be weeded out prior to calling this function.

Set_Visibility:

Narrative: This function receives a flag as its input. The flag is used to specify whether or not the text message is actually displayed on the screen or not. This can be useful for turning on or off a text message. The flag is copied as a Boolean value into the data member Visible.

Diagram:



Interface: This function receives a const int as its input. The integer can be set at one or zero, which determines if the current TEXT_OBJ is visible or invisible. The integer is copied into the data member Visible.

Issues: A TEXT_OBJ that has a default value of 'invisible' will not appear on the screen unless this flag is reset to 'visible'. This may cause some confusion for the user and should be outlined in the help files.

Constraints: Values other than one or zero should be weeded out prior to calling this function.

Get_Dest:

Narrative: This function receives no input. Its duty is to simply return the current TEXT_OBJ's destination coordinate (drawing coordinate) to wherever it was called from.

Diagram:

Interface: This function simply returns the data member Dest_Coord to wherever it was called from. This coordinate is used to draw the text message to the screen. The type returned is a const POINT.

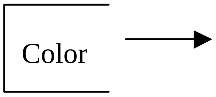
Issues: If the destination coordinate is not defined, the text message may be drawn in odd coordinates on the screen (or may not be drawn at all).

Constraints: The data member Dest_Coord should have a default value of (0, 0). This will ensure that all objects will be drawn (although not at the right place if not specified).

Get_Color:

Narrative: This function receives no input. Its duty is to simply return the current TEXT_OBJ's foreground color to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member Color to wherever this function was called from. The color is used as the foreground color of the text message to be drawn. The return type is a const COLORREF.

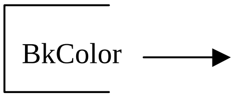
Issues: If the color depth of the game is anything but 16 bit color, Windows will automatically convert the color to fit the specified color space.

Constraints: Default color should be set to (255, 255, 255) so that by default the message will appear white on the screen. This will only be a problem if the background color is specified as white as well.

Get_BkColor:

Narrative: This function receives no input. Its duty is to simply return the current TEXT_OBJ's background color to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member BkColor to wherever this function was called from. The color is used as the background color of the text message to be drawn. The return type is a const COLORREF.

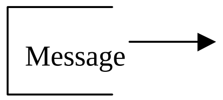
Issues: If the color depth of the game is anything but 16-bit color, Windows will automatically convert the color to fit the specified color space.

Constraints: Default color should be set to (0, 0, 0) so that by default the message will appear on a black background on the screen. This will only be a problem if the foreground color is specified as black as well.

Get_Message:

Narrative: This function receives no input. Its duty is to simply return the current TEXT_OBJ's message to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member Message to wherever this function was called from. The message is the actual string that will be output to the screen. The return type is a const char*.

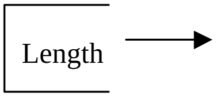
Issues: If the data member Visible is set as false, the text message returned will not actually appear on the screen (actually it will not be drawn at all).

Constraints: None.

Get_Length:

Narrative: This function receives no input. Its duty is to simply return the length of the current TEXT_OBJ to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member Length to wherever this function was called from. The length is used when drawing the text message to the screen. The return type is a const int.

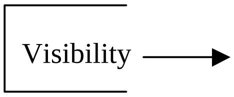
Issues: If the data member Length were to somehow get corrupted or was not set correctly, the message when displayed on the screen could be cut short because the length would imply that the message is shorter than it actually is.

Constraints: None.

Get_Visibility:

Narrative: This function receives no input. Its duty is to simply return the visibility of the current TEXT_OBJ to wherever this function was called from.

Diagram:



Interface: This function simply returns the data member Visible to wherever this function was called from. The visibility flag is used to determine (when drawing) if the text message will appear on the screen or not. The return type is a const int.

Issues: If the data member is set to ‘invisible’, the text, no matter if it should be on the screen or not, will not be drawn on the screen.

Constraints: None.

Get_Trans:

Narrative: This function receives no input. Its duty is to simply return the transparency of the current TEXT_OBJ to wherever this function was called from.

Diagram:

Interface: This function simply returns the data member Transparent to wherever this function was called from. The transparency flag is used to determine (when drawing) if the text message will have a filled background (opaque) or a clear background (transparent). The return type is a const int.

Issues: If the transparency is set to true, the background color that was set will have no bearing on what is actually drawn on the screen. The only time the background color of the TEXT_OBJ will be seen will be when this flag is set to false.

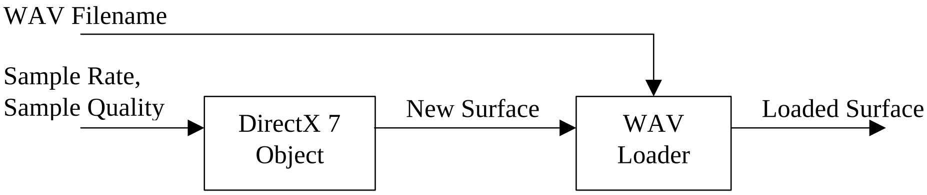
Constraints: None.

SOUND_HANDLER:

Create_Surface:

Narrative: Receives as input a filename, and sample rate and quality. Based on the sample rate and quality, the appropriate DirectX 7 calls are made to allocate memory for surface creation. Once the memory is set, the filename is used to load the specified WAV file onto the DirectSound 7 surface. Once completed creating and loading the surface, the surface is returned.

Diagram:



Interface: The function receives a filename (const char*) and a sample rate and quality (both const int). Using these, the function interfaces with the DirectX 7 Object (DDObjectNew) to create a new DirectSound 7 surface capable of holding a WAV file. Once the new surface has been created, the function opens up the specified WAV file (using filename) and loads the specified WAV onto the new surface. Once complete, the new surface is returned.

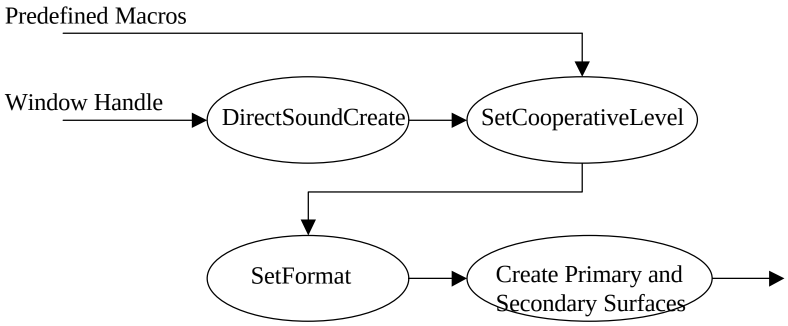
Issues: This function is not capable of determining how much system memory is left available. Assuming that the user loads too many sounds into memory, the program will most likely crash.

Constraints: Illegal sample rates and sample qualities should be discarded and a log file should reflect the problem. In this instance a sample rate of 44MHz should be used and a sample quality of 16 should be used. DirectX 7 will automatically convert the sound of lesser quality to fit these parameters.

DSound_Init:

Narrative: Receives a “hwnd” as input (being the window handle). Using the window handle and several other predefined macros (#defines) found in a separate header file (defines.h), the DirectSound object is first created. Once created the Cooperative level is set (how to share resources with Windows). The Format of the sound buffer will be set and the Primary surface is created. A secondary surface (sound back-buffer) is created for use in streaming sound files and is attached to the primary surface.

Diagram:



Interface: The input to the function is a const HWND. The HWND is the handle to the window that was created in another function. Using the window handle, a DirectSound 7 object is created. The DirectSound 7 Object is the base object for all other sound surfaces to be attached to. After the object is created it can be used for playing sounds using calls defined within the DirectSound 7 object.

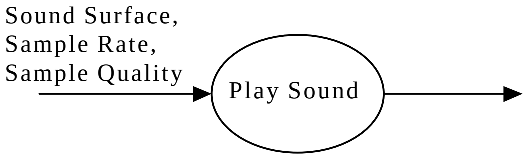
Issues: If the user does not have DirectX 7 installed on his/her machine and their compiler is not linking to the correct DirectX 7 files, this function will not even compile.

Constraints: The GUI builder will have to determine which files are small enough to load into memory, and which files should be streamed from the hard drive.

Play_Sound:

Narrative: Receives a surface, a sample rate and a sample quality. The surface is the surface that will be used for the sound file itself. The surface has a WAV loaded onto it so that using the sample rate and quality sent in, the sound could be played. If the sound clip is too large to fit into memory, the sound clip will be streamed in piece by piece off of the hard drive.

Diagram:



Interface: A DirectSound 7 surface is sent in as well as two const int. The DirectSound 7 surface contains a specific WAV file already loaded onto it. If the sound file is too large to fit into system memory, this function will have to stream the file off of the hard disk.

Issues: The more sounds being played at once, the harder it will be to distinguish one sound from the other. If one of the sounds being played represents an important role in the game, it may be drowned out by the other sounds playing.

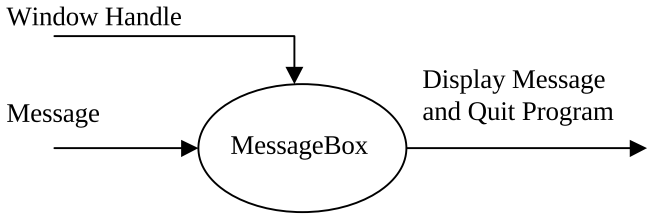
Constraints: Due to the above issue, we might consider allowing a sample pitch to be used to control the loudness of the sound being played.

We should also think about the possibility of streaming all sound files instead of just some sound files. This will most likely be easier to program and in the long run will probably work better.

Error:

Narrative: Error is a very simple function that receives a string and outputs a text box to the screen specifying the error, and shuts down the game.

Diagram:



Interface: The only input is a const char* containing the error message to put on screen. Once on the screen the user has to click on the "OK" button or hit enter. Once clicked, the function posts a quit message and the window callback function kills the application.

Issues: This of course causes the game to crash because it shuts down the program automatically. Another function should be made to have a similar function but not shut down the program (for minor errors).

Constraints: Possibly allow the user to attempt to continue running the program in spite of the error that occurred.

SOUND_OBJ:

Get_Alias:

Narrative: This function simply returns the Alias of the SOUND_OBJ. The Alias is the user given name to the sound surface in question.

Diagram:

Interface: This receives no input and simply sends the Alias back to wherever this function was called from. Return type is a const char*.

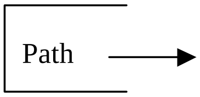
Issues: An attempt to call Get_Alias for a SOUND_OBJ that has no Alias set may crash the program because it would attempt to play a sound that is not created yet. This should not be allowed.

Constraints: This function must check to see if the Alias for the current SOUND_OBJ has been set before trying to send back undefined information.

Get_Path:

Narrative: This function simply returns the path of the SOUND_OBJ. The path is the filename and path of the desired WAV file to load to the sound surface.

Diagram:



Interface: This function receives no input and simply sends back the path the wherever this function was called from. Return type is a const char*.

Issues: If no path yet exists for the current SOUND_OBJ, an empty string will be returned. This path is used in loading the WAV to the sound surface. If the path is NULL, there will be no path to load to the surface and the program may crash.

Constraints: Possibly set a default sound so that the program does not crash if the path has not been set. Or possibly make the sound loading function not kill the program and instead continue processing.

Get_Surface:

Narrative: This function simply returns the sound surface pointer of the SOUND_OBJ. The surface pointer is the pointer to the surface that will be loaded with the specified WAV file from 'Path'.

Diagram:

Interface: This function receives no input and simply returns a surface pointer to wherever this function was called from. Return type is a const LPDIRECTSOUNDSURFACE7.

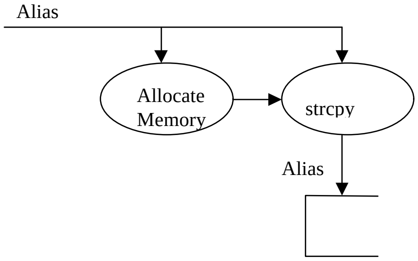
Issues: Any sound effect or music in the game rely upon these surfaces for use in playing the sounds. If the surface was not loaded correctly the game output will either not work at all or not work correctly.

Constraints: None.

Set_Alias:

Narrative: This function receives as input a string. The string is the user-defined name that will represent this sound surface. This is so each sprite defined can be pointed to whichever sound surface needed via their alias.

Diagram:



Interface: This function receive as input a const char* containing the user-defined alias to the current sound surface. First memory is allocated to contain the string, then the string is copied in using strcpy (found in string.h).

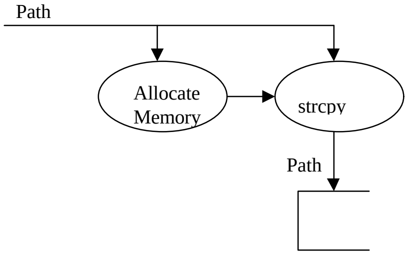
Issues: Resetting an alias may cause problems when using strcpy. The alias may end up being not what was originally sent in.

Constraints: The alias should be kept short, but this will not be enforced, as it will be simpler to just let it be the size it wants to be. Since the alias is user-defined and used within the data files, most users will see that by creating a long alias to begin with just makes them have to type more later on.

Set_Path:

Narrative: This function receives as input a string. The string the filename and path of the filename of the WAV file to be loaded onto the surface. The string is set to the SOUND_OBJ data member Path.

Diagram:



Interface: This function receives a const char* as its input. First enough memory is allocated to contain the string. Then the string is simply set to the current SOUND_OBJ's data member Path. This will be used later to load the surface with the specified filename.

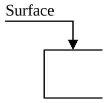
Issues: Resetting a path may cause problems when using strcpy. The path may end up being not what was originally sent in. This may in turn cause problems when loading the WAV file to the surface.

Constraints: The path should contain a .WAV extension. If the path specified does not contain .WAV, then the path is invalid and an error should be output.

Set_Surface:

Narrative: This function receives as input a pointer to a DirectSound 7 surface. This surface will be used to play a specified WAV file when the game logic specifies to do so. The SOUND_OBJ member Surface is simply set to point to the temporary surface sent in.

Diagram:



Interface: This function receives a const LPDIRECTSOUNDSURFACE7 as its input type. The Surface member within the current SOUND_OBJ is simply set to point to the same memory location as that of the surface sent in.

Issues: Users that do not have DirectX 7 installed on their machine will not be able to compile nor run this particular function.

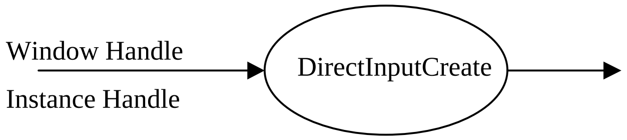
Constraints: None.

INPUT_HANDLER:

DInput_Init:

Narrative: This function receives a window handle and an instance handle. The window and the instance are used to create the DirectInput object. The DirectInput object is what is used by the game to attach input devices to. The input devices (i.e. mouse, keyboard, and joystick) will be used to gather input from the game player to manipulate objects on the screen. If the creation of the object fails, an error message is output.

Diagram:



Interface: This function receives a const HWND and a const HINSTANCE. Both data members are used to create the DirectInput object. The object is created by accessing a few DirectX 7 functions. The object is what all input devices will be attached to.

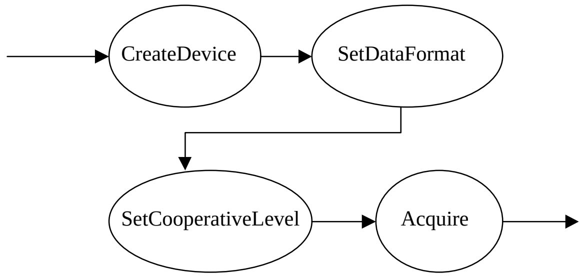
Issues: If the user does not have DirectX 7 installed on their system this function will not compile correctly, nor will it run. It should be noted in the help files that DirectX 7 is needed.

Constraints: The hwnd, and hinstance from the actual window creation will have to be saved and sent into this function for it to work correctly.

Keyboard_Init:

Narrative: This function receives no input. Its duty is to query the keyboard as one of the input devices for use in the game. The keyboard can then be used as an input device to interact with objects in the game.

Diagram:



Interface: This function attaches a keyboard device to the DirectInput object. Once attached the keyboard can be used as input for the game. First the input device is created specifying that the device type is a keyboard. Once created the cooperative level is set. Once that is set, the keyboard is acquired. Now the keyboard is ready for use. If any of the past steps failed, the keyboard will not work correctly or at all during the game.

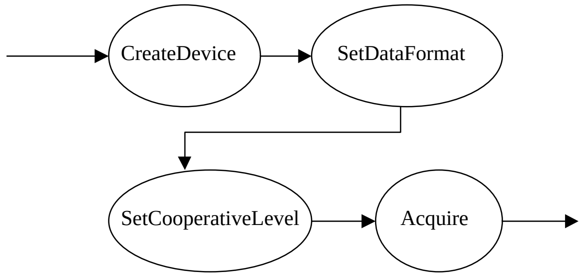
Issues: The user must have the device the game expects attached to the computer prior to running the game or the game will not work correctly.

Constraints: The cooperative level should be set fairly low so that if the user ALT-TAB's out of the game, Windows will regain control of the keyboard. If this were not done its very possible that Windows would lock up.

Mouse_Init:

Narrative: This function receives no input. Its duty is to query the mouse as one of the input devices for use in the game. The mouse can then be used as an input device to interact with objects in the game.

Diagram:



Interface: This function attaches a mouse device to the DirectInput object. Once attached, the mouse can be used as input for the game. First the input device is created specifying that the device type is a mouse. Once created the cooperative level is set. Once that is set, the mouse is acquired. Now the mouse is ready for use. If any of the past steps failed, the mouse will not work correctly or at all during the game.

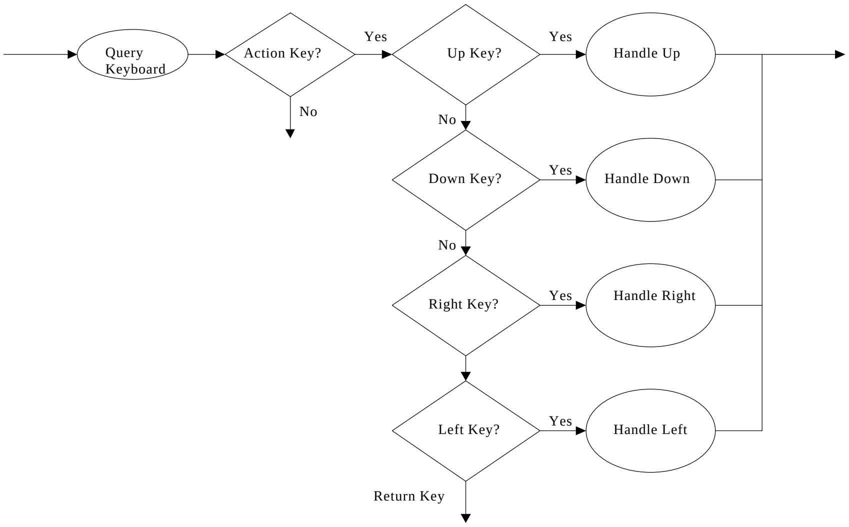
Issues: The user must have the device the game expects attached to the computer prior to running the game or the game will not work correctly.

Constraints: The cooperative level should be set fairly low so that if the user ALT-TAB's out of the game, Windows will regain control of the mouse. If this were not done its very possible that Windows would lock up.

Key_Handler:

Narrative: This function receives the current x and y velocities of the sprite in question. The role of this function is to detect what key is being pressed, and if the key is one of the keys defined by the user as an input key, take the appropriate action. Since there is no way to define all possible actions the user may want the keystroke to represent, only a few actions will be predefined. The predefined actions will consist of strictly moving the object up, down, left, and right respectively. Any other keystroke that has been defined by the user as an important key will be passed back to the main program so its logic can be handled there.

Diagram:



Interface: This function receives two const int as input. The integers are the current x and y velocities. The keyboard will be queried to see what key (or keys) is (are) being pressed at the moment. It will check if the key is one predefined by the user as an action key (defined in the GUI builder). If the key is an action key and represents character movement on the screen, the appropriate calculations will be made using the x and y velocities. If the key being pressed is an action key but does not have a predefined function, the keystroke is returned to the main program so that the keystroke can be handled there.

Issues: These predefined functions for certain keystrokes are necessary. However, the results they produce may not be what the user had planned on.

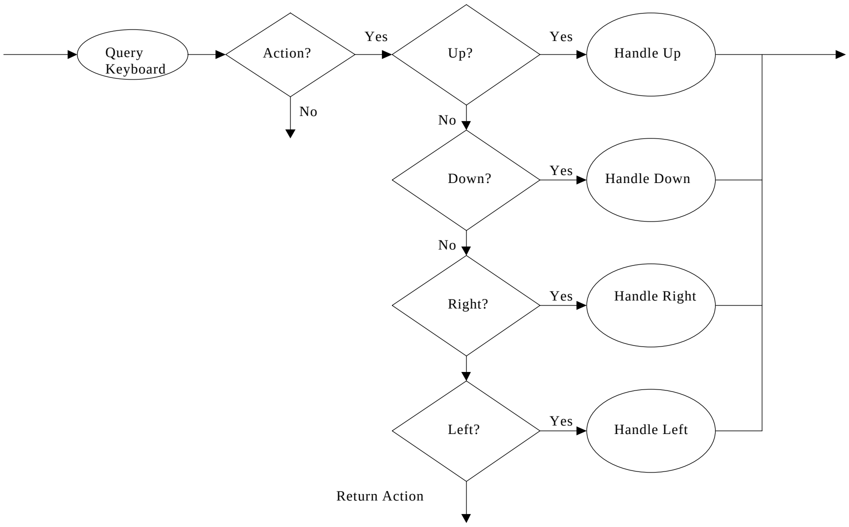
Constraints: Special care has to be taken to allow for GREAT flexibility when defining the actions of the keystrokes.

This will be a fairly large function and should be thoroughly tested.

Mouse_Handler:

Narrative: This function receives the current x and y velocities of the sprite in question. The role of this function is to detect how the mouse is moving and what button(s) is (are) being pressed. If the mouse movement is predefined by the user as an action, the appropriate action must be taken. Since there is no way to define all possible actions the user may want the mouse to represent, only a few actions will be predefined. The predefined actions will consist of strictly moving the object up, down, left, and right respectively. Any other mouse movement that has been defined by the user as an important will be passed back to the main program so its logic can be handled there.

Diagram:



Interface: This function receives two const int as input. The integers are the current x and y velocities. The mouse will be queried to see what movement is happening with the mouse at the moment. It will check if the movement is one predefined by the user as an action movement (defined in the GUI builder). If the movement is an action movement and represents character movement on the screen, the appropriate calculations will be made using the x and y velocities. If the movement is an action movement but does not have a predefined function, the movement is returned to the main program so that it can be handled there.

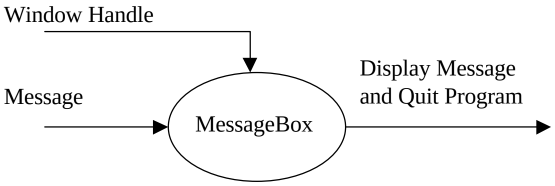
Issues: These predefined functions for certain mouse movements are necessary. However, the results they produce may not be what the user had planned on.

Constraints: Special care has to be taken to allow for GREAT flexibility when defining the actions of the mouse movements. This will be a fairly large function and should be thoroughly tested.

Error:

Narrative: Error is a very simple function that receives a string and outputs a text box to the screen specifying the error, and shuts down the game.

Diagram:



Interface: The only input is a const char* containing the error message to put on screen. Once on the screen the user has to click on the "OK" button or hit enter. Once clicked, the function posts a quit message and the window callback function kills the application.

Issues: This of course causes the game to crash because it shuts down the program automatically. Another function should be made to have a similar function but not shut down the program (for minor errors).

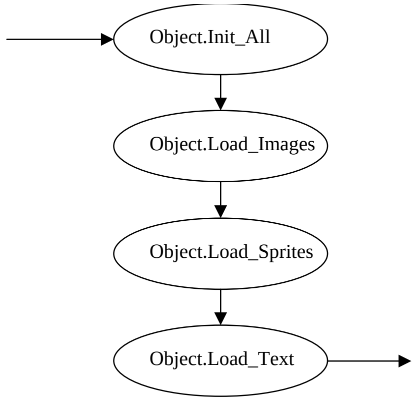
Constraints: Possibly allow the user to attempt to continue running the program in spite of the error that happened.

Main Program:

Game_Init:

Narrative: This function receives no input. Its duty is to wrap all initialization functions into one function call. In most games this will never change. This is the function where all sprites, text messages, and surfaces, are loaded. This means that the data files that contain the information needed by the program to create the sprites, text messages, and surfaces is specified here. Any other things that might need initializing (like initializing DirectDraw, etc.) are also put in here.

Diagram:



Interface: This function receives no input. All game initialization routines are called from within this function. First, DirectDraw, DirectInput, and DirectSound are all initialized. Once this is complete, the sprites, text messages, and surfaces are all initialized. These are initialized by sending in data file filenames to the functions that will know how to read the data and use that data to create the game objects. This function is called once per game.

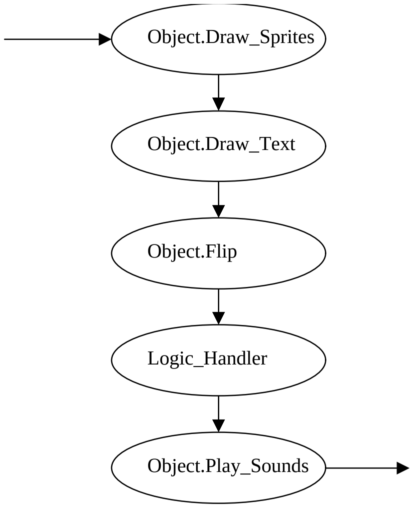
Issues: Undefined data files will most likely cause the application to crash. This should be outlined in the help files so that the user knows to only include valid data files.

Constraints: DirectDraw, DirectInput, and DirectSound MUST be initialized prior to initializing any of the game objects. If the reverse were to occur, the application would crash.

Main_State:

Narrative: This function receives no input. This is the main game loop. The game will loop into this function and repeat every call within so long as the game is running. Any function that manipulates the objects on screen is put in here. The function calls for drawing, playing sounds, handling game logic, and handling user input are all within this function. This allows for the game to be constantly updated until the game is over.

Diagram:



Interface: This function receives no input. The function is called repeatedly and as often as possible (though the frame rate will be limited to 30 frames per second). The first function call is to draw all the sprites and text messages on the screen that should be on the screen. Once this is done, the surfaces are flipped so that the user will see the next frame of animation. After this is done, the game logic functions are called. These will update the screen positions of all objects (also the player based on input). This will also call the collision handler to make sure that no objects are overlapping, and will trigger any events associated with the objects that are colliding. Once complete, and sounds or music that need playing are played. At this point the time is checked and the function sits here until 30 milliseconds have passed. What this does is ensure that if the game is run on a vastly faster machine than it was designed for, the game speed will not change and will remain at roughly 30 frames per second.

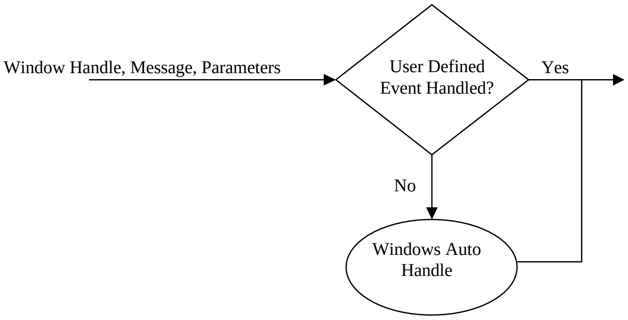
Issues: This function does not ensure that on old machines (slower machines) that the frame rate will achieve 30 frames per second. This is simply set up so that faster computers will not exceed 30 frames per second.

Constraints: The drawing and page flipping should be done first and as immediately as possible. This is to ensure that the animation runs smoothly.

Message_Handler:

Narrative: This function receives the window handle, a message, and two parameters. Based upon the message and the parameters, this function acts accordingly. For a simple example, if the player hits ALT-TAB on the keyboard, the event WM_ACTIVATEAPP will be sent immediately to this function. A global variable is then set to be one of the parameters sent in. If the parameter is false, this means the user wishes to return to windows, and unlock the primary surface and back-buffer so that Windows will regain control of the screen. Any user event (or others) can be handled here, however most often we chose to bypass this routine and handle the event manually.

Diagram:


Window Handle, Message, Parameters
User Defined Event Handled?
Yes
No
Windows Auto Handle

Interface: This function receives a HWND, a UINT, a WPARAM, and a LPARAM. The function is an event handler. The UINT is the message generated by a keystroke or automatically by Windows. Based on the UINT message, the appropriate actions take place. Once this is done, the function returns out, and then returns to the exact place that the program jumped out of the handle this message.

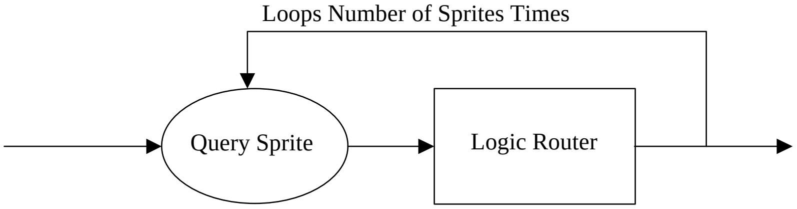
Issues: Windows will automatically handle Any Windows message that is not defined. Most messages will never be explicitly handled by our program and will let Windows act appropriately.

Constraints: Most predefined Windows handler functions are suitable for use in any application. If there is no need to specifically make a change or addition to any occurring event, it’s suggested to let Windows handle it automatically.

Logic_Handler:

Narrative: This function receives no inputs. Its duty is to query a specific object's attributes to see what kind of game logic it is defined to have, and call a specific function for handling that logic. This will be done as many times as there are objects.

Diagram:



Interface: This function receives no inputs. Within the function is a for loop that continues until all sprites (onscreen or off-screen) are accounted for. During each iteration of the loop, the current SPRITE_OBJ is queried to see what attributes have been set for it. Based upon these attributes, logic functions are called to manipulate the sprite's position on screen. For example, if the current sprite is a moving sprite (meaning another creature in the game) and is a solid sprite (meaning when its collision area is overlapping with another this object will stop) and is affected by gravity (meaning this object is drawn downward) then the appropriate logic functions will be called. In this instance the calls go to the function that handles gravity, followed by the collision detection function. If a collision is found, the sprite stops moving downward, if not it continues.

Issues: If the sprite being tested has no attributes defined for it, most of the logic functions will have no affect on that sprite. This should be outlined in the help files.

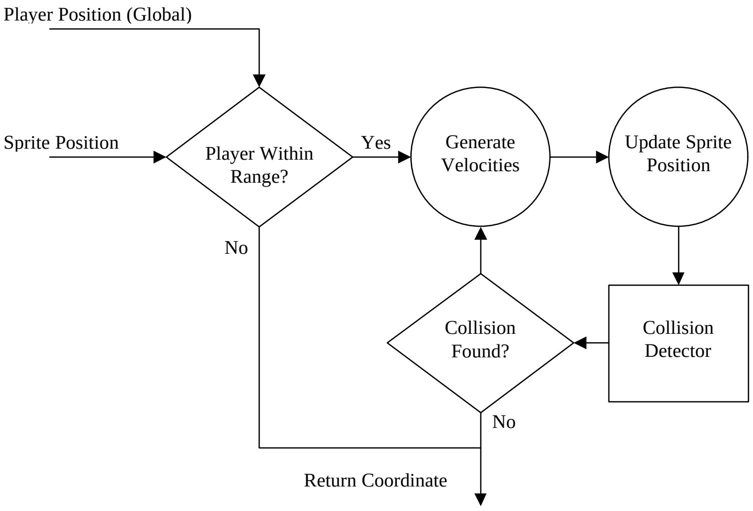
Constraints: Moving sprites (creatures) will have to be tested against every other sprite and collision area for the current level. The more creatures there are, the slower the game will move. It should be considered to put a creature limit for each level.

If the logic function calls are not called in the correct order, undesirable effects may occur. Special care should be taken when determining which logic function has priority. The function should also be thoroughly tested to ensure that it works without any problems.

Logic_Follow:

Narrative: This function is a simple AI routine used solely by other creatures. Static sprites will not have use for this function. This function receives the X and Y position of the current sprite. Using the X and Y position (of the current sprite) and the X and Y position of the player character, the current sprite will be instructed to head towards the player character. Once determined where to move, an X and Y velocity are calculated and the X and Y position's of the sprite are updated. From here the collision detection algorithm is called to determine if that 'movement' caused the sprite to collide with any other object. If not, the function returns. If so, velocities are re-calculated and tested again.

Diagram:



Interface: This function receives a const POINT as input. The POINT contains the X and Y position of the current sprite. These values are tested against the player character's X and Y positions and it is determined which direction the current sprite will have to move to intercept the player. Once determined, an X velocity and Y velocity will be calculated. The velocities will be used to update the sprites X and Y position. The new X and Y position's will be tested with the collision detector function to determine is the movement caused a collision. If a collision is not detected the new position is valid and the function exits. If a collision is detected, new velocities are calculated bearing in mind what direction caused the collision. This is sort of a 'bump and turn' routine. A direction is tested, if it works then exit, if not change the direction.

Issues: None.

Constraints: 'Closed' areas may create some difficulty in generating the correct movement for the sprite. Older arcade games used a similar routine to have creatures hunt the player. Their solution was to specify to the level designers that no closed areas are allowed for the level. This should be outlined in the help files since the GUI builder user is developing the level they should be aware of this.

Another method to fixing this problem would be to allow the creature to back up. This may or may not be more difficult than the previous solution, and both ideas should be tried and tested.

Logic_Flee:

Narrative: This function is a simple AI routine used solely by other creatures. Static sprites will not have use for this function. This function receives the X and Y position of the current sprite. Using the X and Y position (of the current sprite) and the X and Y position of the player character, the current sprite will be instructed to move away the player character. Once determined where to move, an X and Y velocity are calculated and the X and Y position's of the sprite are updated. From here the collision detection algorithm is called to determine if that 'movement' caused the sprite to collide with any other object. If not, the function returns. If so, velocities are re-calculated and tested again.

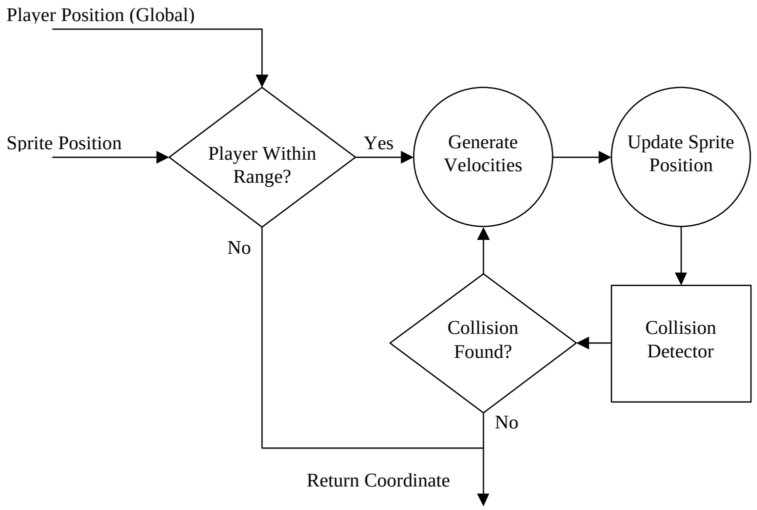


Diagram:

Interface: This function receives a const POINT as input. The POINT contains the X and Y position of the current sprite. These values are tested against the player character's X and Y positions and it is determined which direction the current sprite will have to move to avoid the player. Once determined, an X velocity and Y velocity will be calculated. The velocities will be used to update the sprites X and Y position. The new X and Y position's will be tested with the collision detector function to determine is the movement caused a collision. If a collision is not detected the new position is valid and the function exits. If a collision is detected, new velocities are calculated bearing in mind what direction caused the collision. This is sort of a 'bump and turn' routine. A direction is tested, if it works then exit, if not change the direction.

Issues: None.

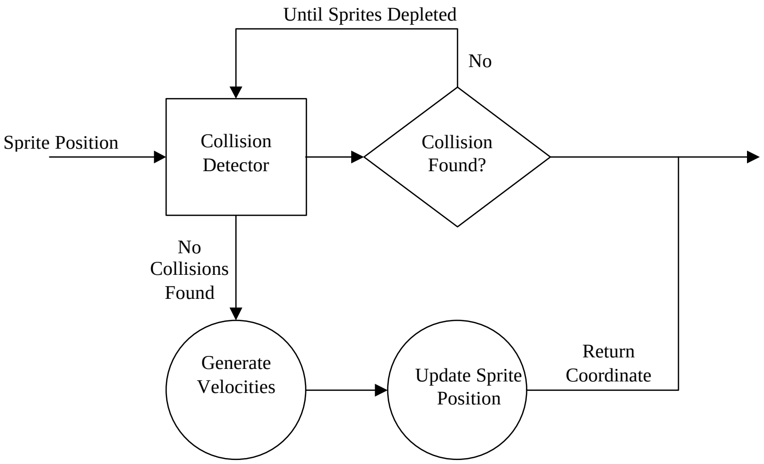
Constraints: 'Closed' areas may create some difficulty in generating the correct movement for the sprite. Older arcade games used a similar routine to have creatures hunt the player. Their solution was to specify to the level designers that no closed areas are allowed for the level. This should be outlined in the help files since the GUI builder user is developing the level they should be aware of this.

Another method to fixing this problem would be to allow the creature to back up. This may or may not be more difficult than the previous solution, and both ideas should be tried and tested.

Logic_Gravity:

Narrative: This function is a simple AI routine used solely by other creatures. Static sprites will not have use for this function. This function receives the X and Y position of the current sprite. Based on that position the collision detection function will be called and will determine whether or not the current sprite is already colliding with a solid object from the top. If so, the function exits. If not, using the Y coordinate sent in and a predefined constant for gravity, the new Y position of the sprite is calculated.

Diagram:



Interface: This function receives a const POINT as its input. The POINT contains the X and Y coordinates of the current SPRITE_OBJ. Using these coordinates, the collision detection function is called to determine if the current SPRITE_OBJ is already colliding with another solid SPRITE_OBJ, from the top. If a collision is detected, the function exits with no change to the SPRITE_OBJ's position. If a collision is not detected, the Y coordinate is updated by a predefine velocity that represents the acceleration due to gravity.

Issues: If the collision detection algorithm is not working correctly, sprites may continue falling until the fall right off of the screen. Also, sprites may stop falling even if they are not colliding with another sprite.

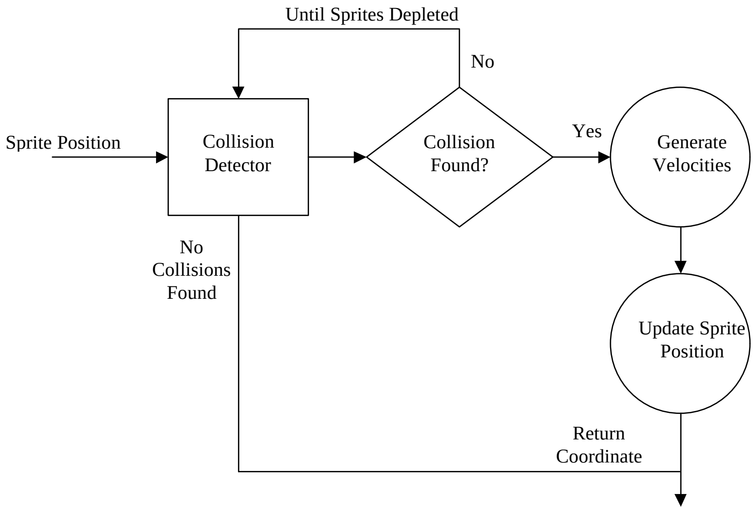
If the acceleration due to gravity is too high, upon the next game loop the sprite may pass clear through solid objects (sprites that are too small may be overlooked).

Constraints: Sprites will most likely have to contain a 'mass' data member. Using the mass, it can be calculated how fast the sprite should accelerate downward. This can be done without using a mass data member simply be limiting the acceleration due to gravity to a constant. This may make more sense.

Logic_Bounce:

Narrative: This function receives the X and Y coordinate and velocity of the current sprite. Based on the position and the velocity, a new position for the sprite is determined. This position is tested for a collision in the collision detection function. If there is not collision, then the function exits. If there is a collision, depending on the direction the collision was caused from, the velocity will be reversed. Once reversed, the sprite's position is updated, and the function exits.

Diagram:



Interface: This function receives a const POINT and a const int as its input. The POINT contains the X and Y position of the current SPRITE_OBJ. The integer is the relative velocity. Based on the coordinate and the velocity, a new X and Y position is calculated. Once calculated, the collision detection algorithm is called to see if the new coordinate caused a collision. If not, the function exits. If so, a new coordinate is calculated (based on the negative velocity and the direction that caused the collision). Once the new coordinate is calculated, the function exits.

Issues: If the collision detection algorithm is not functioning correctly, this may falsely detect a collision and reverse the current sprite for no reason. It may also not detect an actual collision and the sprite would pass through a solid object.

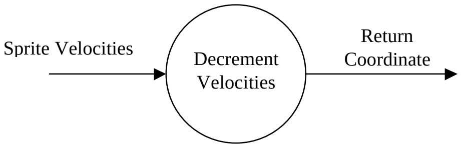
If somehow the sprite in question were to get caught inside of another sprite's collision area, the current sprite would continue to bounce back and forth within the object.

Constraints: None.

Logic_Friction:

Narrative: This function receives the X and Y velocity of the current sprite. Using these velocities, a new velocity is generated. Meaning that the velocity returned from this function will be smaller than the velocity sent into the function. This is so that during each game loop, the sprite in question will slow down and finally come to a stop. The new velocities are returned.

Diagram:



Interface: This function receives two const int as input. The integers represent the X and Y velocities of the current SPRITE_OBJ. New velocities are calculated (which will be smaller in value than those sent in). Each game loop the velocities get smaller and smaller until they finally hit zero. Once at zero, the SPRITE_OBJ comes to a complete stop. The new velocities are returned.

Issues: If a sprite's friction flag is not set, this function will not have any bearing on the sprite's movement. The sprite will simply never stop moving until the game is over.

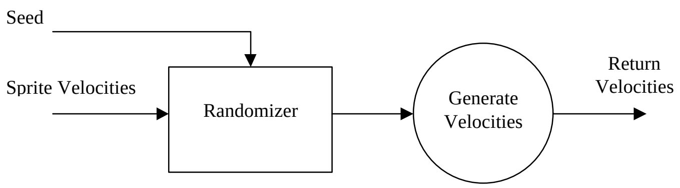
This function will also have no affect on the player sprite so long as the player is using the input device to control the sprite. Once the player stops using the device, the player sprite will eventually come to a stop (assuming the player sprite's friction flag is set).

Constraints: This should be a fairly simple function. The real physics equation for calculating friction could be used, but on large levels with more creatures, this may cause the game to take a serious performance hit.

Logic_Random:

Narrative: This function receives the X and Y velocities of the current sprite and a ‘seed’ for the randomizer. The seed is used more as a delimiter for the randomizer. It will limit the actual value spit out by the randomizer to be below that number. Using the randomizer function, a new X and Y velocity are calculated and are returned from this function.

Diagram:



Interface: This function receives three const int as input. The first two integers represent the current SPRITE_OBJ’s X and Y velocities. The last integer is the number to use when calling the randomize function to limit the output of the randomizer. The value returned by the randomizer is set to the X and Y velocities and they are returned.

Issues: If the seed sent in for use in the randomizer is very large, the current sprite will most likely jump off of the screen. This is because a very high number could be set to be the X or Y velocity, and the sprite’s position after the next game loop would be somewhere off the screen.

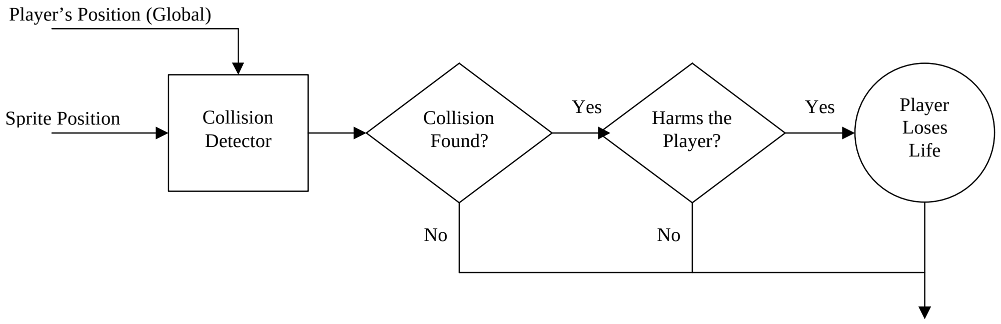
If the sprite’s random flag is not set, this function will have no bearing on the sprite’s position.

Constraints: Negative numbers for the randomizer seed are not acceptable. If a negative number is sent in to the randomizer, it will be converted to its absolute value instead.

Logic_Kill_Player:

Narrative: This function receives the X and Y position of the player sprite. The collision detection function is called to see whether or not the player is colliding with any other sprites. If so, those sprites are queried to determine if they will harm the player. If the sprite does not harm the player, then the function exits. If the sprite does harm the player, update the player 'alive' state (a global variable). If the player alive state is zero, during the next game loop, the player will lose a life. If the player's lives are all lost, the game is over.

Diagram:



Interface: This function receives two const int as input. The integers are the X and Y position of the player SPRITE_OBJ. The player's position is tested to see if it's colliding with any other SPRITE_OBJs. If the player is colliding with another sprite, test that sprite to see if it harms the player. If so, decrement the global variable 'alive' so that the appropriate action can be taken next game loop.

Issues: If the sprite that the player is colliding with does not have its solid flag set, there will be no way to detect if that sprite is colliding with the player sprite. So regardless of whether the sprite should harm the player, it will not.

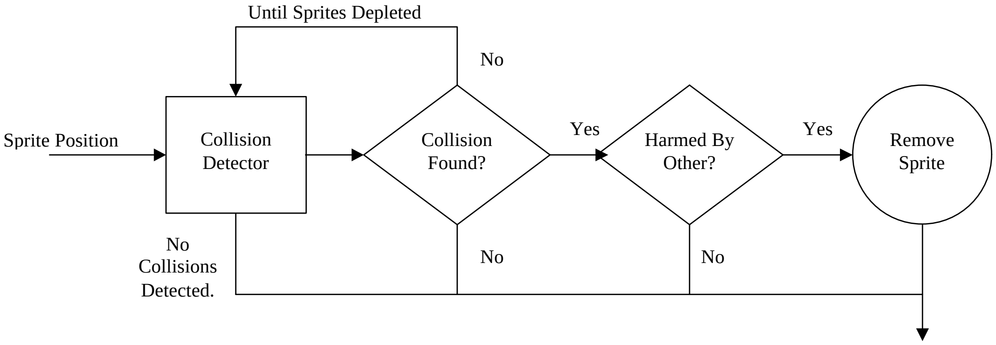
Constraints: It might be considered to first query all sprites to determine which actually could harm the player. Once they are determined, call the collision detection function to see if a collision occurred. This may speed the algorithm up a bit.

It may also be considered to destroy the sprite instead of simply hiding it from view. This will free up memory and speed up certain processes (collision detection, etc.).

Logic_Killed_By_Other:

Narrative: This function receives the X and Y position of the sprite. The collision detection function is called to see whether or not the current sprite is colliding with any other sprites. If so, those sprites are queried to determine if they can be killed by other sprites. If the sprite is not set to be harmed by another sprite, then the function exits. If the sprite is sensitive to other sprites, simply set the sprite's visible flag to false, and its solid flag to false. This is so that the sprite no longer 'exists' to the player.

Diagram:



Interface: This function receives two const int as input. The integers are the X and Y position of the current SPRITE_OBJ. The sprite's position is tested to see if it's colliding with any other SPRITE_OBJs. If the current sprite is colliding with another sprite, test that sprite to see if it harms the current sprite. If so, hide the current sprite by setting its visibility flag to false, and set the sprite's solid flag to false as well (so that the user cannot interact with is by accident).

Issues: If the sprite that the current sprite is colliding with does not have its solid flag set, there will be no way to detect if that sprite is colliding with the current sprite. So regardless of whether the sprite should harm the current sprite, it will not.

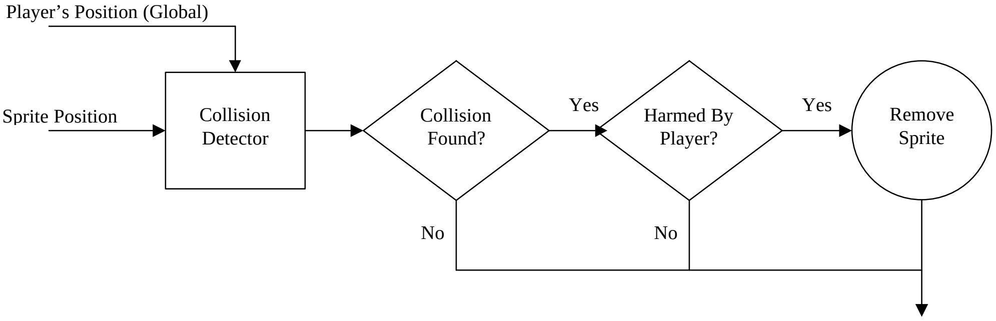
Constraints: It might be considered to first query all sprites to determine which actually could harm other sprites. Once they are determined, call the collision detection function to see if a collision occurred. This may speed the algorithm up a bit.

It may also be considered to destroy the sprite instead of simply hiding it from view. This will free up memory and speed up certain processes (collision detection, etc.).

Logic_Killed_By_Player:

Narrative: This function receives the X and Y position of the current sprite. The collision detection function is called to see whether or not the player is colliding with this sprite. If so, the current sprite is queried to determine if the player will harm it. If the player cannot harm the sprite, then the function exits. If the player can harm the sprite, simply set the sprite's visible flag to false, and its solid flag to false. This is so that the sprite no longer 'exists' to the player.

Diagram:



Interface: This function receives two const int as input. The integers are the X and Y position of the current SPRITE_OBJ. The player's position is tested to see if it's colliding with this SPRITE_OBJ. If the player sprite is colliding with the current sprite, test that sprite to see if the player can harm it. If so, hide the current sprite by setting its visibility flag to false, and set the sprite's solid flag to false as well (so that the user cannot interact with is by accident).

Issues: If the sprite that the player sprite is colliding with does not have its solid flag set, there will be no way to detect if that sprite is colliding with the player sprite. So regardless of whether the player should harm the current sprite, it will not.

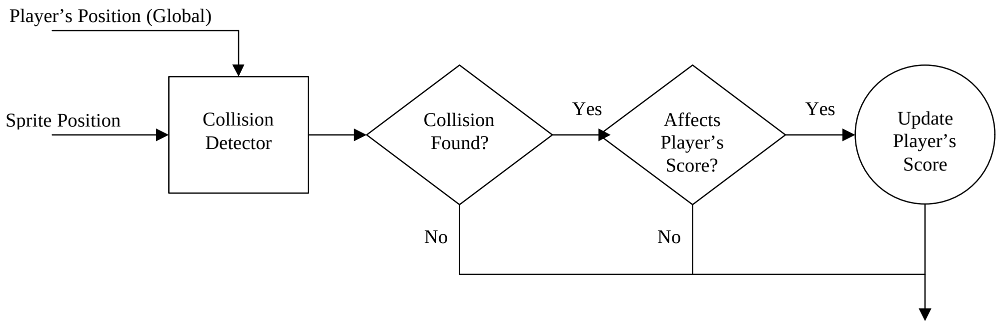
Constraints: It might be considered to first query all sprites to determine which actually could harm other sprites. Once they are determined, call the collision detection function to see if a collision occurred. This may speed the algorithm up a bit.

It may also be considered to destroy the sprite instead of simply hiding it from view. This will free up memory and speed up certain processes (collision detection, etc.).

Logic_Score:

Narrative: This function receives the X and Y position of the current sprite. The collision detection function is called to see whether or not the player is colliding with this sprite. If so, the current sprite is queried to determine if the player's score will be affected by it. If the players score is not affected then the function exits. If the player's score is affected, the player's score is updated (by some fixed amount). Most likely this sprite will also be eliminated but that will not always happen.

Diagram:



Interface: This function receives two const int as input. The integers are the X and Y position of the current SPRITE_OBJ. The player's position is tested to see if it's colliding with this SPRITE_OBJ. If the player sprite is colliding with the current sprite, test that sprite to see if the player's score is affected by it. If so, update the player's score.

Issues: If the sprite that the player sprite is colliding with does not have its solid flag set, there will be no way to detect if that sprite is colliding with the player sprite. So regardless of whether the player's score should be affected by colliding with the current sprite, it will not.

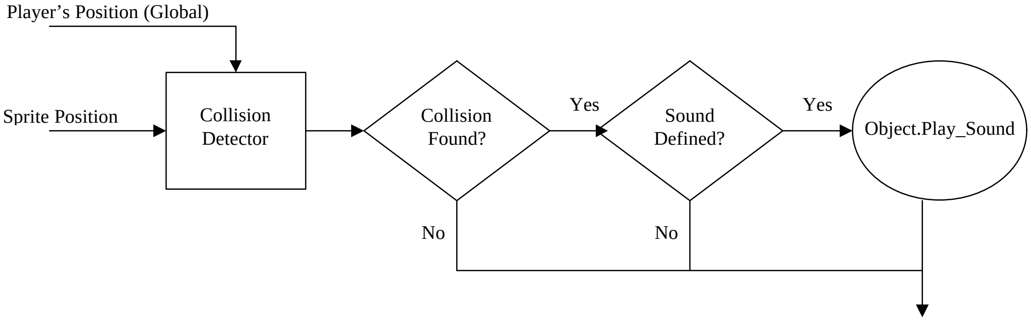
Constraints: It might be considered to first query all sprites to determine which actually could harm other sprites. Once they are determined, call the collision detection function to see if a collision occurred. This may speed the algorithm up a bit.

It may also be considered to destroy the sprite instead of simply hiding it from view. This will free up memory and speed up certain processes (collision detection, etc.).

Logic_Sound:

Narrative: This function receives the X and Y position of the current sprite. Based on these positions, the collision detection function is called to determine if the player is colliding with the current sprite. If so, the sprite is queried to see if the sound index has been set. If it has been set, the sound that the index corresponds to will be played.

Diagram:



Interface: This function receives a const POINT as its input. The POINT is the X and Y coordinate of the current SPRITE_OBJ. The collision detection function is called with respect to this coordinate to determine if the player is interacting with this SPRITE_OBJ. If the player is colliding with the current sprite, the sprite is queried to see if the sound index has been set. If the sound index has been set, the sound corresponding to that index will be played. The function then exists.

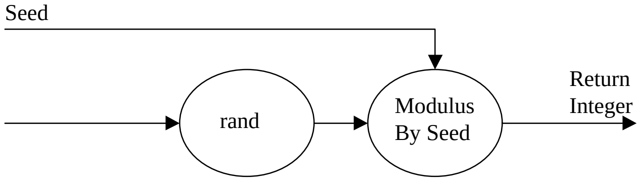
Issues: If the sprite that the player sprite is colliding with does not have its solid flag set, there will be no way to detect if that sprite is colliding with the player sprite. So regardless of whether a sound should be played when colliding with the current sprite, it will not.

Constraints: It might be considered to first query all sprites to determine which actually could harm other sprites. Once they are determined, call the collision detection function to see if a collision occurred. This may speed the algorithm up a bit.

Randomizer:

Narrative: This function receives a seed for its input. The seed is used to limit the number returned from this function. The function makes a call to rand (defined in stdlib.h). That function generates a large positive random number. The seed sent in to our function will take the modulus of that number with the seed to cut down the range of the number. The new number is returned.

Diagram:



Interface: This function receives a const int as its input. The integer is used to limit the range of the random number generated by the rand function (rand is found in stdlib.h). Using the modulus operation with regard to the integer sent in cuts down the number returned from rand. The calculated number is returned out of the function as a const int.

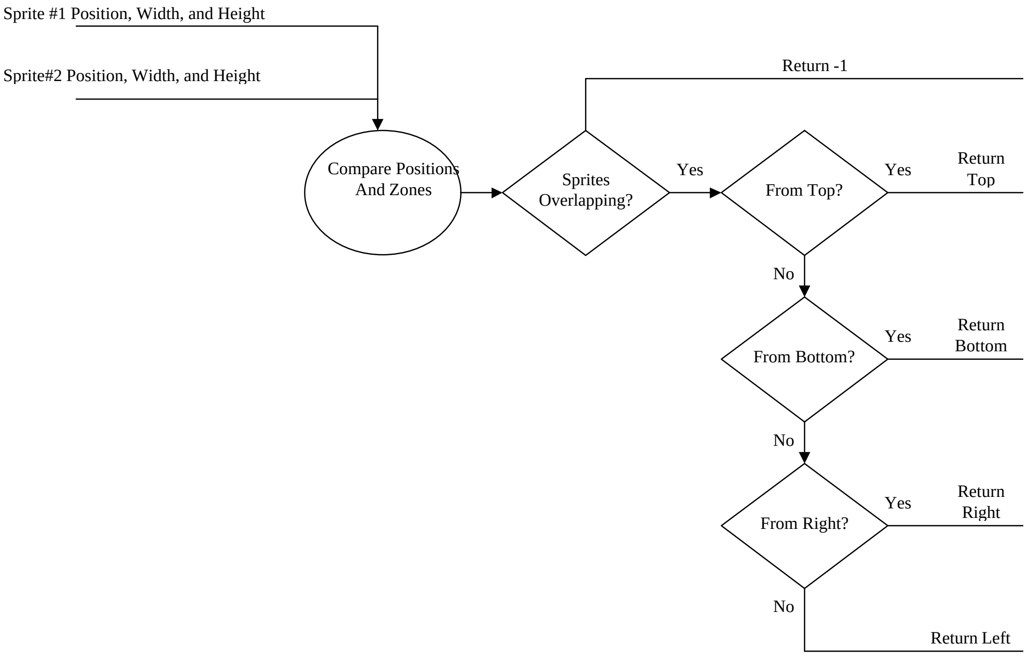
Issues: None.

Constraints: None.

Collision:

Narrative: This function receives two X and Y coordinates two widths and two heights. The first coordinate is the position coordinate of the current sprite being tested. The second it the position coordinate of the sprite its being tested against. Based on these coordinates, and the width and height for each sprite, this function will determine if the two objects are colliding. If they are colliding, a predefined constant is returned representing what direction the collision occurred from.

Diagram:



Interface: This function receives two const POINT, and four const int as input. The two POINTs are the position coordinates of the SPRITE_OBJ being tested and the SPRITE_OBJ its being tested against. The four integers are the width and height values of each SPRITE_OBJ. Using these values, this function will test if the first SPRITE_OBJ is overlapping with the second SPRITE_OBJ's collision area. If it is overlapping, a constant representing top, bottom, left and right is returned. The return type is a const int.

Issues: If the collision zones are very small, and the velocity of the current sprite is very large, the sprite may pass right through another sprite. This is because the current velocity makes the current sprite step right over the sprite its being tested against.

Constraints: There definitely should be a way of determining what side of the object the collision took place on. In some games a collision from the top kills the enemy and a collision from the side kills the player. This is impossible to determine unless the direction of the collision is returned.

Software Interface Description

External Machine Interfaces

We have no plans to support any external machine interfaces.

External System Interfaces

We have no plans to support any external system interfaces.

Human Interface

The interface that we have chosen to use will be designed in Microsoft Visual Basic Version 6. The interface will utilize many of the common controls included in Visual Basic, and the majority of windows-based applications. The interface will be a graphical user interface that provides the user with a quick and easy way to position objects and design and build a two-dimensional game. For a more detailed description of the interface, please refer to the User Interface Design section found below.

User Interface Design

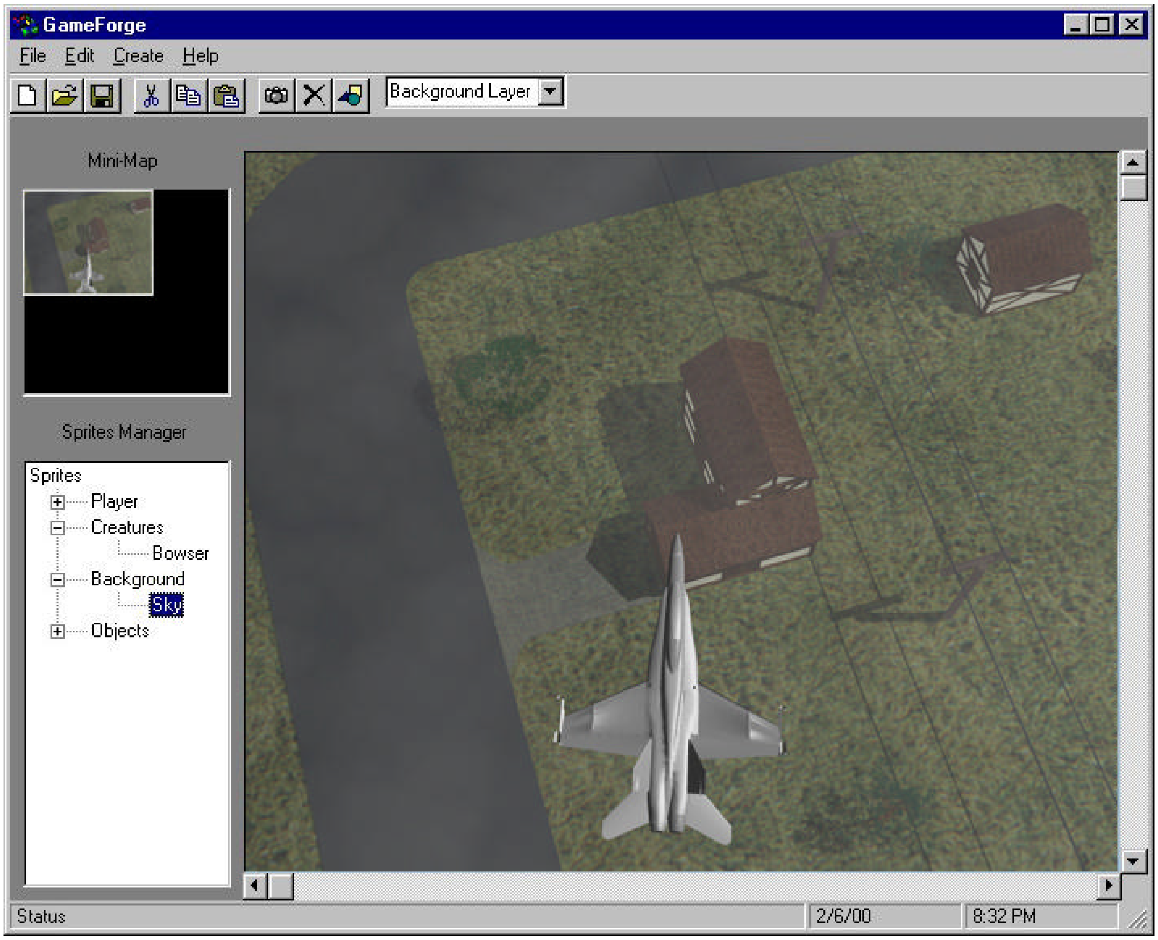
Description of the User Interface

Description

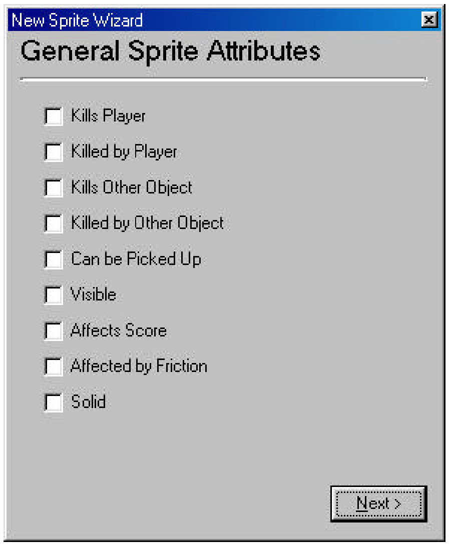
The interface will have on the left side of the screen, a treeview control, which displays elements in a directory tree structure. Placed in this treeview will be categories with which the sprites that user has input will be categorized. The user will be able to click on these categories and see the expanded list of the sprites that are under that particular category. In addition the user will be able to click on a particular sprite and bring up all of that sprite's properties. When the sprite wizard is up, the right side of the screen will be the sprite's image, and an area where that image can be placed on the screen, onto any of the particular backgrounds that the user has chosen. There will be a standard menu bar that is present on nearly every Microsoft Windows application, along with a toolbar for quicker access to the commands embedded under the various menu options.

Screen Images

The Level Editor:



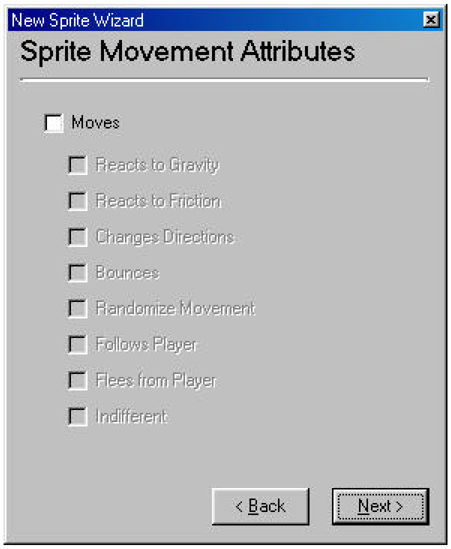
New Sprite Wizard: Attributes



General Sprite Attributes

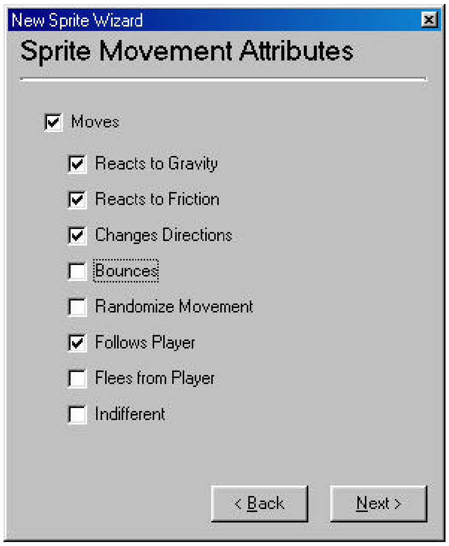
□ Kills Player
□ Killed by Player
□ Kills Other Object
□ Killed by Other Object
□ Can be Picked Up
□ Visible
□ Affects Score
□ Affected by Friction
□ Solid

New Sprite Wizard: Movement



Sprite Movement Attributes

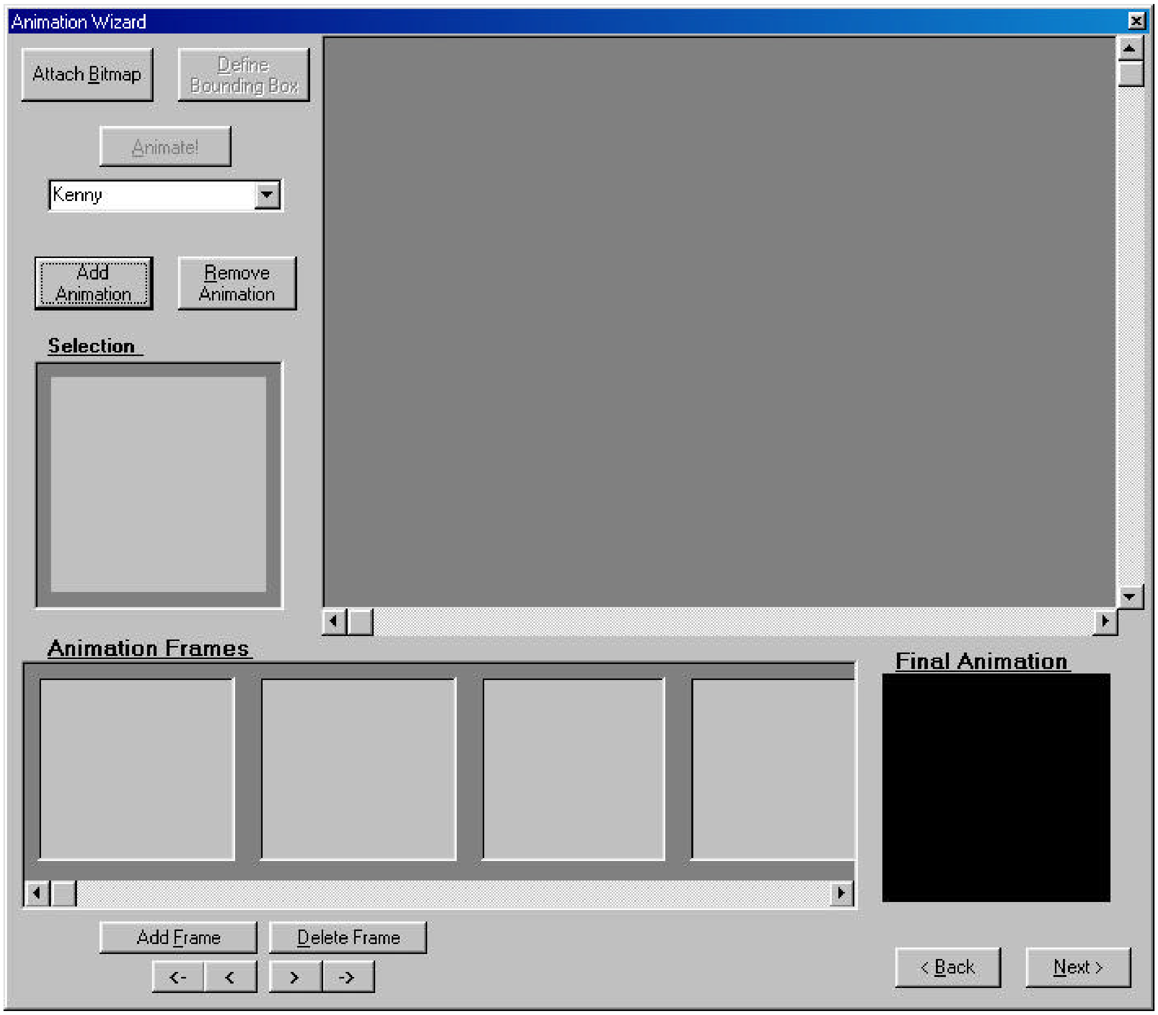
□ Moves
- □ Reacts to Gravity
- □ Reacts to Friction
- □ Changes Directions
- □ Bounces
- □ Randomize Movement
- □ Follows Player
- □ Flees from Player
- □ Indifferent



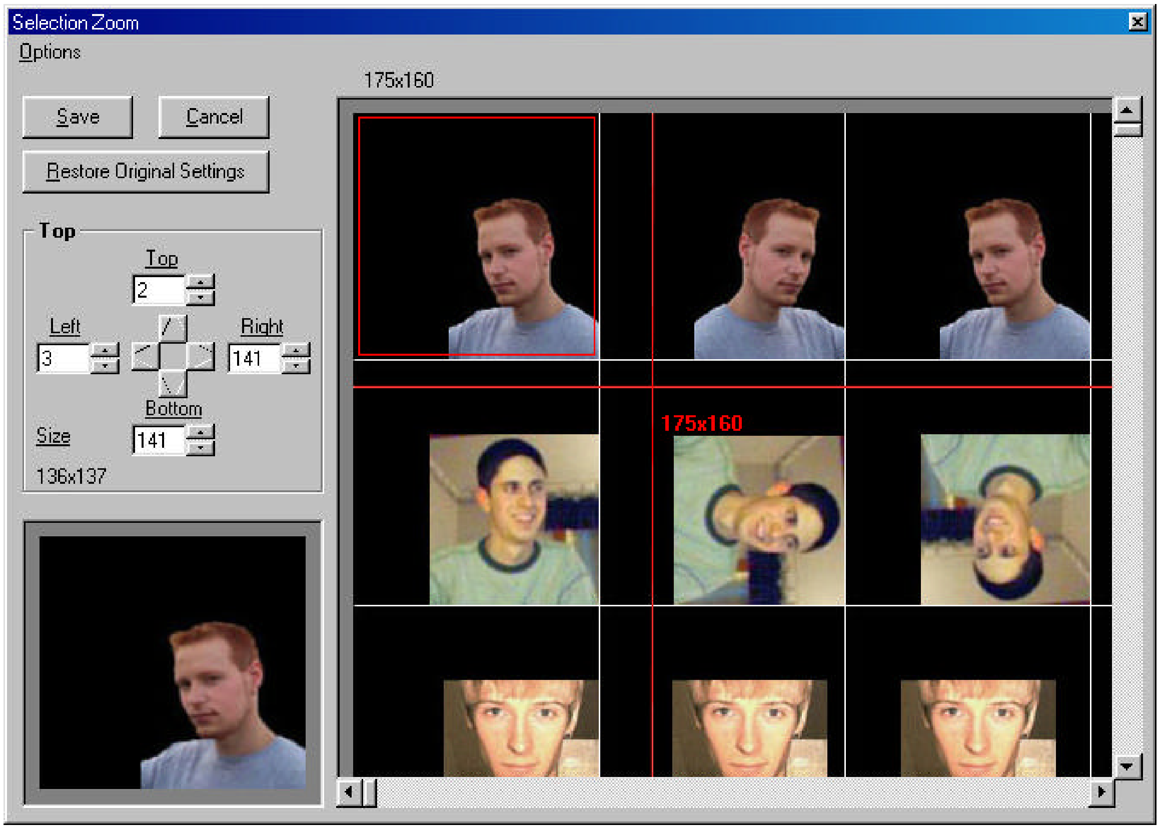
Sprite Movement Attributes

☑ Moves
- ☑ Reacts to Gravity
- ☑ Reacts to Friction
- ☑ Changes Directions
- □ Bounces
- □ Randomize Movement
- ☑ Follows Player
- □ Flees from Player
- □ Indifferent

New Sprite Wizard: Animation main screen



New Sprite Wizard: Animation - Select BMP



Top

Top: 2

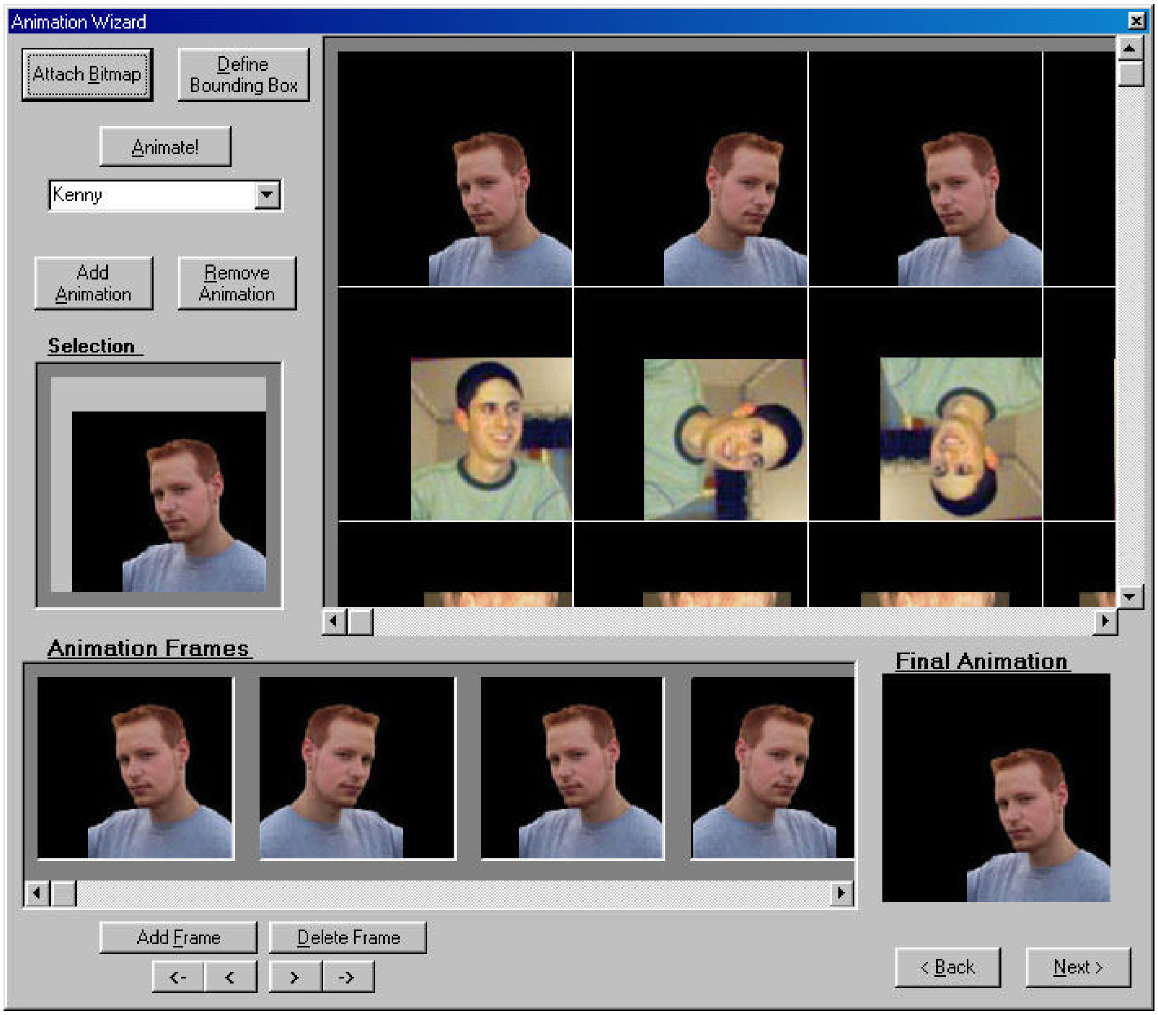
Left: 3

Right: 141

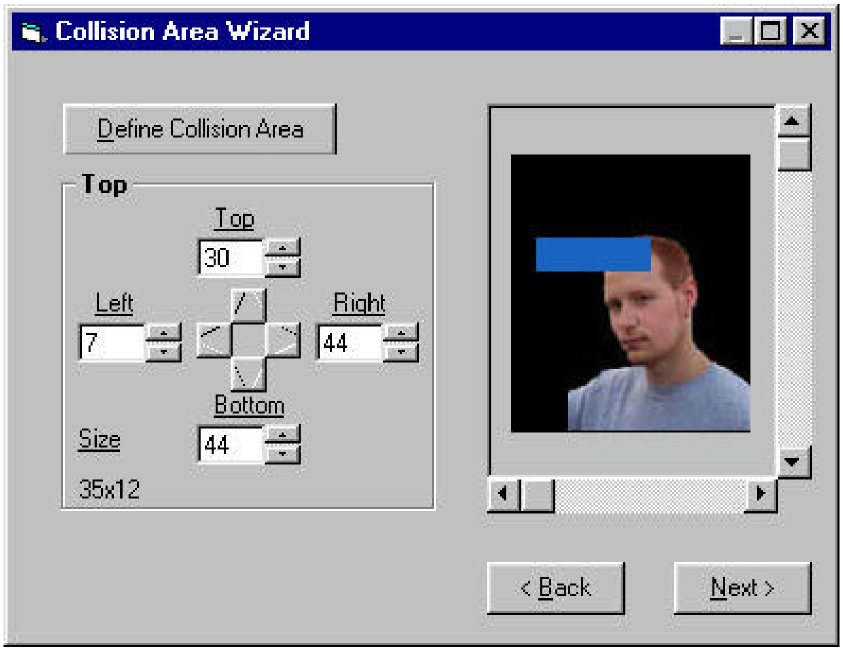
Bottom: 141

Size: 136x137

New Sprite Wizard: Animation Main Screen (with BMP)



New Sprite Wizard – Set Collision Area:



Top

Top 30

Left 7

Right 44

Bottom 44

Size

35x12

Objects and Actions

Treeview Control:

The Treeview Control will be used to allow the user to see what sprites are currently loaded into the application. The sprites will be categorized and be divided into different branches that can be expanded or collapsed. When a particular user selects one of the loaded sprites, the user will be able to position the image associated with that sprite.

MiniMap:

The MiniMap will allow the user to see what portion of the overall level currently being edited is in view. It will also allow the user to move quickly to any portion of the level, regardless of level size.

Surface Selector:

The Surface Selector is a combobox that will allow the user to display only the surfaces of the type that is selected in the combobox. This is useful when designing the level because there may be multiple surfaces on the screen in a particular area. To be able to edit/view individual objects, it will be necessary to have the surface of that type selected.

Interface Design Rules

Eight Golden Rules of Dialog Design

1. Strive for consistency
2. Short cuts for frequent users
3. Design dialogs to yield closure
4. Offer information feedback
5. Offer simple error handling
6. Permit easy reversal of actions
7. Support internal focus of control
8. Reduce short term memory load

Data Display Guidelines

1. Consistency of display
2. Efficient assimilation of information by the user
3. Minimize memory load
4. Compatibility between data entry and display
5. Flexibility of user control

Menu Guidelines

1. Shallow, wide menus preferred over tall deep menus
2. User has access to all relevant items without referencing a manual
3. Logical item presentation sequence
    a. Numeric
    b. Alphabetic
4. Icons are harder to recognize than text during visual search
5. Don’t assume the user will notice cues like color or border changes
6. Allow key presses for frequent users
7. Ensure consistent navigation

Screen Formatting Guidelines

1. Focus on readability and user acceptability
2. Don’t clutter the screen (white space is free)
3. Choose pleasing color combinations
4. Use full width of the screen
5. Keep only relevant info on the screen
6. Use device codes to move cursor, clear lines, etc.

Guidelines for Effective Use of Color

1. Use color to group similar items
2. Use colors with connotation if reasonable (red means stop, etc.)
3. Limit the total number of colors (7 +/- 2)
4. Watch out for bad color combinations (red/blue, blue/black)
5. Keep in mind people may view it in monochrome

Prompt and Response Guidelines

1. Use upper/lower case letters (for emphasis as needed)
2. Tasks should be interruptible without loss
3. Give user a means of controlling multiple screens
4. Allow user to verify and edit responses
5. Give user a reasonable amount of time to respond
6. Compose in screen rather than lines
7. When long delays are inevitable, put up an indicator

Components Available

See the MSDN Library for Microsoft Visual Basic 6.0 for a complete list of all components available.

UIDS description

Microsoft Visual Basic 6.0 is a programming language that allows the user to quickly create complex applications for Windows, without all of the overhead required using other languages. It allows the user to pick from a list of practically thousands of controls, and draw them on the screen. These controls then have certain events, methods, and properties that can be set. When a particular event fires, the code associated with that event is executed. See the MSDN Library for Microsoft Visual Basic 6.0 for a complete list of all components available, including each component's properties, methods and events.

Restrictions, Limitations, and Constraints

Performance/Behavior Issues

GameForge is designed to be compatible with the Microsoft Windows 9x operating system. Microsoft Windows NT 4.0 and earlier versions will not be supported (Windows NT only supports Microsoft DirectX up to version 3.0. DirectInput had not been implemented at this time, making this version of DirectX very limited.) Microsoft Windows 2000 should also be compatible, but is not directly supported.

GameForge also requires Microsoft DirectX 7.0 or above. Users may also want to obtain the DirectX 7.0 SDK if they plan on expanding the GameForge library files beyond their original scope.

GameForge also requires the Microsoft Visual C++ 6.0 compiler. GameForge’s VC++ code may be compilable using Borland or some other VC++ compiler, but functionality is not guaranteed.

Program Limitations

GameForge does not directly support two players. This includes two player simultaneous and two player turn based.

GameForge supports only limited mouse support. Users will only be able to control player movement. Users will not be able to select specific sprites to gain control over that sprite.

GameForge only supports 800 by 600 resolution and 640 by 480 resolution, both at 16 bit color depth.

GameForge only supports WAV files for use in sound effects. Any other type of audio file (i.e. MIDI, MP3, etc.) will have to be converted to WAV format for use in GameForge.

Testing Issues

Classes of Tests

Unit Testing

Individual engine components (Draw Handler, Sound Handler, etc.) will be tested separately. Do to the GameForge engine's modular design, there is no need for test beds. All components can be tested through the Object Handler.

The interface will be unit tested in three parts:

- Level editor
- Wizards
- File exporting

All unit testing will be done in White Box fashion.

Integration Testing

Combined engine components will be tested as a whole. To maintain maximum control over the testing criteria, all data files will be made specifically for testing purposes.

The level builder will be tested to ensure proper communication between the interface and the database.

High-Order Testing

The High-Order testing will be performed on the complete, integrated system. "In-house" beta testing will take place at this time and PA Software staff members will attempt to construct fully functional games using GameForge.

Public Beta Testing

Parties outside PA Software will be asked to help with the final testing phase. Each beta tester will be given a copy of the software, and the preliminary help files (these will not be completed until immediately prior to GameForge's final build.) Beta testers will be expected to submit bug reports and any opinions they may have concerning the software (especially the interface layout.)

Expected Software Response

Interface

Some issues are expected for more complicated game designs.
Flawless exporting to engine.

Engine

Fair performance on PII and better machines
Errorless compilation of code.
Errorless/Low error execution of code without alteration.

Performance Bounds

General Hardware Requirements

Processor: PII 200 MHz
System Memory: 32 MB RAM
Video Memory: 4 MB VRAM

Interface Performance Requirements

Robust help and tutorial files: The windows based help files must be detailed enough to assist users, but written clearly enough to maximize understanding, especially for novice users. They must be as complete as possible, to assist advanced GameForge users seeking specific solutions. Tutorials must be clear and complete, and must not assume anything of the user. Graphical cues may be helpful. The GameForge help system must include a menu structure for more casual searching, as well as a competent search engine too allow users access to specific data.

Flawless exporting to engine: All files exported from the level builder must be free of errors. Any errors generated at this point may result in code that is not executable.

Engine Performance Requirements

Avg. FPS: Games utilizing the GameForge engine must retain an average number of frames per second (number to be finalized at a later time) using the minimum requirements.

Mem usage: Due to the costly memory usage of uncompressed bitmaps and wave files (which GameForge uses), unused memory must be freed properly once it is no longer being used. File paging from the hard drive is unacceptable (if the user has the minimum system requirements.)

Errorless compilation of code: There should be zero errors when compiling code that has been exported from the GameForge level builder. This only includes code that has been compiled immediately after being exported from the level builder. It is the user's responsibility to care for their code after they have altered it.

Errorless/Low error execution of code without alteration: Some ambitious game designs may not run perfectly or with all features intact after being exported from the GameForge level builder. However, the code should still be executable with reasonable results.

Identification of Critical Components

Interface

Main Level Editor – This is the main interface, and displays a graphical representation of the game/level a user is designing. A tree-view of all created objects is also represented here. All wizards and other functions can be accessed from this interface.

Input Wizards – There are a number of wizards provided to guide the novice user through the necessary steps for game development. They range from sprite generation, to game logic, to input devices. The wizards interact directly with the user interface.

Exporting GameForge Files (.gmf) – Files are stored with a unique extension used exclusively by the GameForge system. These files are similar to .cpp files but will not be compilable. They are intended as a temporary storage during game creation. They are generated by the user interface.

Exporting VC++ Files (.cpp) – Finished projects can be saved as .cpp files that can be compiled with Microsoft's Visual C++ compiler to create an executable file for the game. The VC++ engine runs these files.

Engine

Object Handling: The Object handler is the 'hub' of the engine, and controls all data flow through the program.

Draw Handling: The Draw Handler controls all image processing. All surfaces are created here.

Sound Handling: The Sound Handler controls all sound processing. All sound is done in a streaming fashion.

Input Handling: The Input Handler controls all input processing. All user defined movement controls are defined here.

Logic Handling: The Logic Handler controls all game logic. Sprite attributes are passed into the logic handler, and are interpreted based on world attributes. This includes collision detection, gravity, pathing, etc.

Appendices

Requirements Traceability Matrix

Requirement: Interface easy to use by novices and advanced users.
Design: The interface has been designed for maximum functionality, but with the user's comfort in mind.

Requirement: Creature AI reasonably advanced.
Design: The new Logic Handler includes pathing and attribute functions that simulate creature AI.

Requirement: Exported code from the level builder is immediately compilable.
Design: All exported code will be compilable, and most code will be executable as well (for less complicated game designs.)

Packaging and Installation Issues

Installation Package

GameForge will utilize Wise Installer 8.0 from Wise Solutions, Inc. as its software installation package. Wise Installer was chosen because of the design team's familiarity with the software. Licensing negotiations are still underway, and should they fall through, PA Software currently retains a software license for InstallShield.

Installation of GameForge will be completely automated, and appropriate Program Groups and Icons will be created as per the user's request. GameForge will also include an uninstall feature, to remove itself from any local machines.

Product Manual

GameForge will ship with a detailed user manual, covering the following topics:

- Installation / Removal
- Introduction to the software
- General usage instructions
- A tutorial detailing the creation of a simple game, from start to finish

The GameForge software package, in conjunction with the website, will include an extensive list of help files and tutorials. The manual is not meant to be a replacement for these facilities, especially where troubleshooting and question lookup are concerned, but is instead intended to assist the user in quickly familiarizing themselves with GameForge.

Product Packaging and Distribution

GameForge will ship with a visually pleasing CD jewelcase, containing the GameForge logo, and "at a glance" information about GameForge. The manual will fit nicely into the CD jewelcase cover. The manual and packaging artwork are only available when ordering from the GameForge website, for a nominal fee.

GameForge will be distributed via the GameForge website, where it will be available for free download. The software itself will have a splash screen that can only be bypassed after an undetermined number of seconds. This screen can be bypassed if the copy of GameForge is licensed via the GameForge website, for a nominal good-will donation.

Design Metrics to Be Used

PA Software will utilize two design metrics during the GameForge development process:

- Function Point
- CoCoMo

These will be used in estimating costs and development time. For more detailed information regarding these estimates, see the Software Project Plan document.

Supplementary Information

Additional Information can be found at:

GameForge website: www.patheticattempts.com/gameforge
Pathetic Attempts website: www.patheticattempts.com