SOFT351

Assignment 1

Joseph Kellaway

# Version Information

## Campus

Microsoft Visual Studio Professional 2015 – Version 14.0.25431.01 Update 3

DirectX SDK June 2010

## Home

Microsoft Visual Studio Enterprise 2015 – Version 14.0.25431.01 Update 3

DirectX SDK June 2010

# End User Guide

## Part 1

The user is presented with a 3rd person perspective of a winged-bear which they can move around the screen. The arrow keys are used to rotate the bear and alter the pitch that it is angled at. The W and S keys are used to add momentum to the bear both forwards and backwards respectively. When turning at near maximum speed the bear will lean into the turn to mimic the animal shifting its weight. To prevent the user from losing the bear off-camera, the F4 key can be pressed to toggle a camera mode that follows the bear. The spacebar can be pressed to make the bear roar. The F1 key can be used to toggle whether these commands are shown on-screen whilst using the program. When angled skyward, the bear can use W and S to fly. Whilst flying; the wings flap, a flapping sound is made periodically and the bear always leans whilst turning. If neither momentum key is pressed when flying, the bear glides whilst slowly losing momentum and falling to the ground. If the bear is angled towards the ground, it straightens upright when it lands.

Whilst moving and turning the bear, you should also notice shadows and highlights forming on it, as well as a large shadow of the bear on the floor moving relative to it.

## Part 2

The user is presented with a revolving pig. It is loaded from an “.obj” file and displayed on the screen. To change the pig to another shape, change the file loaded at line 494. To change the texture applied to the shape that is loaded, change the file loaded at 583.

# Programmer’s guide

## Part 1

Global variables that are relevant and the same for all objects are set up. This includes all of the objects required for setting up the DXUT environment that the bear program runs in. Global variables added to the Microsoft example code starts at line 78 with:

* Meshes (to render later), the projection matrix.
* A diffuse only pixel shader for loading objects that won’t display shadows correctly.
* Boolean flags to detect keyboard input.
* A variable to alter the rate at which the skybox spins.
* Angles (in radians) that are level with the ground and the y-value of the ground itself.
* Speed increment due to gravity (currently incremented per frame, so an arbitrary scale to reduce it is included) – used for falling.
* Air density for slowing down objects in the air. A value that represents earth’s air pressure at 20 degrees Celsius.
* A value to raise the camera above the bear when in follow mode, with a stabiliser value that scales based on the angle of the bear to reduce camera movement when the Y-direction is changed.
* A “Bear” – an object class designed to hold all of the attributes and actions specific to what the bear can do.
* Light colours and an arbitrary shininess value (used when doing per-pixel reflection effects).

Thing3D is currently used as an abstract class of variables extended by the Bear class. It has been set up to include variables required for every item in the world to be an instantiation of it (currently the skybox and floor). It contains the following variables with the means to retrieve and set them from the main class:

* Location, rotation and scale variables and a vector to hold the initial direction.
* Matrices for calculating how changes to these variables should work and be represented.
* Pointers to shaders, buffers, the mesh and a struct for holding world view projection and world matrices (passed to the shader) so that the object can render itself (not currently in use due to an error passing the ID3D11InputLayout pointer). Pointers are used so that all Thing3Ds can share the same shaders rather than have shaders for every single item in the world (as this would make adding items inefficient for performance).

Thing3D also holds constructor and destructor methods as well as methods to prepare and render itself, though these are currently not in use due to the issue with the ID3D11InputLayout pointer.

The Bear object is a Thing3D with extra functionality. It holds the following additional variables with the means to retrieve and set them from the main class:

* Current speed and fall speeds to indicate how fast the bear is moving.
* Maximum forward and reverse speeds to stop the user from giving the bear too much momentum.
* A drag coefficient and mass used when calculating how much the bear should be slowed.
* A tilt value in radians for whilst turning.
* Maximum climb and descent values in radians to prevent the horrible interaction with going beyond the middle of the ceiling/floor.
* A current and resting position for the wings (in radians), used for calculating the angle to display the flapping wings at.

The Bear class handles the following functionality when requested by the main file:

* Turning (manipulation about the x-axis to alter where the bear is facing):
  + It offsets the current rotation about the x-axis by a value multiplied by the number of frames turn is held for.
* Tilt left/right (manipulation about the z-axis to make it look as if the bear is “leaning” in to a turn):
  + It offsets the current rotation about the z-axis by a value multiplied by the number of frames turn is held for up, up to the “max tilt” point indicated by the previously assigned variable. When turning ceases, straighten up returns the bear to the global horizontal position for the z-axis in the same manner so that it doesn’t “snap” back into position.
* Forward and reverse:
  + Offsets the current speed by a value multiplied by the number of frames the button is held for. Pre-defined variables representing maximum speeds cap this.
* Air resistance:
  + Uses modified Newtonian mechanics (I don’t calculate an arbitrary “size” value for the bear, I let the coefficient handle it) to calculate how much momentum should be reduced by. This is only called when the user is not applying momentum to the bear.
* Friction:
  + Reduces current speed by 1% each time it is called. This is called once per frame when bear is allowed to reach the ground.
* Gravity:
  + Increases the current fall speed by the gravity fall speed increment before calculating the bear’s new y-value. If the bear would fall through the floor it places the bear on the floor and sets the bear’s speeds to 0.
* Flapping wings:
  + Changes the wing’s rotation about the z-axis to a sin value related to the time divided by an arbitrary value. This function then plays a flapping sound. When flapping is to stop (i.e. when the bear is on the ground or forward momentum is ceased), the bear will flap up to one more time to get the wing back into the resting position.
* Tilt up/down (increase/reduce elevation if momentum is added):
  + It offsets the current rotation about the y-axis by a value multiplied by the number of frames turn is held for. Levelled out in the same way if the bear “crashes” into the floor face-first.
* In air:
  + Simply compares the bear’s current y-value to the globally determined ground value.
* Roar:
  + Play a wav file of a roaring sound.
* Move (increase/decrease momentum):
  + Merges the bear’s current rotation values into the math library’s rotation matrix, calculates the current direction based on this matrix and the bear’s “spawn” direction and then normalizes the value. Calculates an area just behind to get the “rear” camera to follow and applies speed in the current direction before setting the new X, Y and Z coordinates and finally returning the “rear” vector for the camera.

Next the structs are set up to pass variables to the shaders and DXUT is used to set up the view window. Each frame render, the window clears itself before:

* Calling the bear’s move method (to retrieve a vector to position the rear camera).
* Setting up the view (eye, at and up vectors):
  + This can be toggled between a spawn view and a view behind the bear’s current position.
* Calculating the bear’s translation and scale matrices before multiplying them with the already-calculated rotation matrix to form the bear world matrix.
* Calculation the world view projection matrix by multiplying the bear world matrix with the view and projection matrices.
* Setting up a light vector.
* Setting up constant buffers to pass to shader files and update the immediate context as necessary.
* Setting the pixel shader to the main pixel shader responsible for drawing meshes with diffuse, specular and ambient reflection all added to the mesh.
* Rendering the bear mesh.
* Setting up wings’ scale, translation and matrices, and multiply them by the bear world to make them move relative to the bear, then render them.
* Setting up the floor scale and translation matrix and render it. This time we don’t include the bear because we don’t want the floor to move relative to the bear.
* Setting up a floor vector to project shadows on to. This is combined with the light vector to create a shadow matrix. It then creates a shadow translation using the bear’s current coordinates minus a small arbitrary value.
* Multiplying each of the bear parts world matrices to create a “shadow world” for them and then multiplying these shadow worlds with the view and projection matrices to make shadow world view projections. Render these with the appropriate meshes.
* Creating a translation matrix for the skybox using the eye vertex from the view matrix so that the viewer is always in the center of the skybox, and if on follow cam, the bear can never escape it either. Creating a rotation matrix about the y-axis based on the time and an arbitrary multiplier to give the effect that the clouds are moving before scaling the skybox down with a scaling matrix (because this one is huge). Multiplying these three matrices together to create the sky world and add the view and projection matrices as before.
* Before rendering apply a different shader to the immediate context so that only diffuse light is considered. This is to prevent the skybox from becoming darkened since with a light-vector taken into account, it would be in permanent shadow. Render the mesh.

The renderer takes a Boolean parameter to indicate whether it is rendering shadow or not, and it skips over a line of code to load the mesh texture variables if it is, leaving the shadow black.

DXUT is set up to handle keyboard inputs. Relevant keyboard inputs for the bear’s controls flag global Booleans as true, which in turn are monitored with other conditions such as whether the bear is on the ground or not and the program calls repositioning calculations as required.

Finally, when the window is closed memory to DXUT items, pointers, meshes and the bear is deallocated to minimize memory leaks.

## Part 2

The mesh loader is directed to a “.obj” mesh to load, it then loads the mesh by reading lines in the file and setting up vertices, vertex textures, vertex normals and faces from the file.

It reads the “.obj” file line-by-line and identifies information by reading the first letters as the following:

* Vertices - “v “.
* Vertex textures – “vt”.
* Vertex normals – “vn”.
* Faces – “f”

Upon finding the item, it extracts the relevant values and it places them into a vector of them that grows to whatever size the mesh needs it to be.

It then counts the number of faces found, creates a mesh with that many faces and populates its positional, texture and vector normal data and generates an indices list to ensure that pairs the right values together properly.

After this it reads the “.mtl” file in the same manner as the “.obj” file, looking for the following items:

* Specular exponents – “Ns”.
* Ambient values – “Ka”.
* Diffuse values – “Kd”.
* Specular values – “Ks”.
* “Ni” values – “Ni”.
* “D” values – “d”.
* Illumination models – “illum”.

Much like with the “.obj” file, it extracts the relevant values and places them into a vector that grows to whatever size the “.mtl” file requires.

It then passes the vertices over the DXUT for rendering and cleans up the variables used for the loading the shape and texture data.

After this it loads the texture file (a “.jpg”) to map onto the loaded shape and texture data when it renders. It then enters a loop of rendering the image once per frame, with a rotate about the origin.

# Additional Note to the Programmer

## Part 1

There has been an issue where occasionally an exception is thrown (see appendix 1) on startup. The exception declares that there is an access violation regarding memory location 0xFFFFFFFF and freezes at a line in xnamathvector.inl. Simply running the program again fixes the issue (or doesn’t, but repeatedly restarting will do so eventually). I have attempted numerous methods to fix this issue which is linked to the program not allocating 16-byte aligned memory to XMVECTORs.

I have found a solution which is to change the settings of the debugger to run on a 64-bit platform which defaults to 16-byte aligned memory rather than 8-bytes in 32-bit (see appendix 2). If this causes issues on your machine, please revert the Platform column in the screenshot provided, but expect to see the aforementioned exception occurring on occasion.

# Evaluation

## Part 1

Firstly, I am proud that I have successfully implemented two shaders, allowing objects other than the skybox to keep their specular and ambient lighting whilst allowing the skybox to maintain its intended colours rather than the dark shadow that occurs if these lighting types are applied to it with a light source from outside. I am also proud that I have improved upon the base lab by applying a shiny effect on the bear, to imitate the shimmer of the fur when the light hits it.

I am proud of the physics used in the prototype – I used Newtonian laws as a basis for them where possible. I would have preferred to have been more accurate when calculating how wind resistance would slow the bear, by including actual bear dimensions, but I did manage to get a reasonable value for the mass included.

I am also quite pleased with the way that the lean looks when the user turns the bear with the left and right keys whilst applying momentum, but would improve this by applying science to the angle, rather than the current method which is an arbitrary angle between upright and a maximum, based on how long the bear has been turning for.

During turning implementation, I did experiment with the in-built quaternion calculations, but this was after applying the previously mentioned leaning effect, and couldn’t get quaternions to look correct with it included. My quaternion code is still within the bear’s move method, but commented out as when facing the floor and turning, the bear doesn’t move in a way that looks natural due to the lean. To prevent gimbal locking with the standard rotation calculations, and to prevent a very disorientating camera change when going beyond perpendicular to the floor, maximum climb and descent angles were used, which convinces the user that the movement is reasonably natural.

Getting the skybox to translate relative to the viewer was relatively easy once scales had been considered, though when combining it with a simple rotation matrix to give the effect of moving clouds is very visually satisfying.

At first glance the bear’s shadows (complete with flapping wings) looks believable, though during testing and upon closer inspection it is apparent that shadow mapping isn’t quite natural. The shadow only meets the bear at the spawn point, so if the bear isn’t in the air or at spawn then it reveals itself as inaccurate.

I have handled floor collision well in most circumstances. The bear loses any momentum that might send it below the floor if travelling forwards and it pitches up if it is facing downwards, and the bear won’t reverse if it is on the ground and reverse is angled downwards. However, there is a scenario where reversing whilst airborne that the user can manage to get the bear under the floor. Due to the collision calculations involved with the floor, the only thing the user can do from here is apply forward momentum if the bear is angled higher than horizontal, so the issue isn’t huge, but it is something that I would like to improve upon.

The largest change I would make to the prototype if continuing or repeating the project would be to stop using frame time for physics. Currently, the physics look pretty good in the Babbage labs (at 200-400fps) because I have managed to fine-tune the numbers involved to suit those frame rates. However, at home I regularly achieve 2000-2500 fps, which replaces the glide effect of a creature with wings and gradually incrementing fall rate, with an almost instantaneous stop just before the bear hits the floor within a second of the momentum buttons being un-pressed. I have searched through DXUT objects to attempt to force a maximum frame rate without success, though this would still be unreliable. Ideally, I would want to have a fixed measurement, such as a timer, handling the intervals for the gravity and friction physics.

Upon writing up this document it has come to my attention that I haven’t moved the translation and scaling matrices to calculate within the Bear class yet, which is something else I would amend if given more time.

I am fairly happy with the deallocation of memory performed within the project, but if given more time I would like to spend a considerable portion of it ensuring that there are no memory leaks present within the code.

Finally, the light direction vector is currently set up inside the frame rendering loop every single time it renders. If improving or repeating creation of the prototype, I would instantiate this as a global variable and then alter this within the frame rendering loop by rotating the light vector around the eye in some way to cause the shadows to suggest the passage of time.

## Part 2

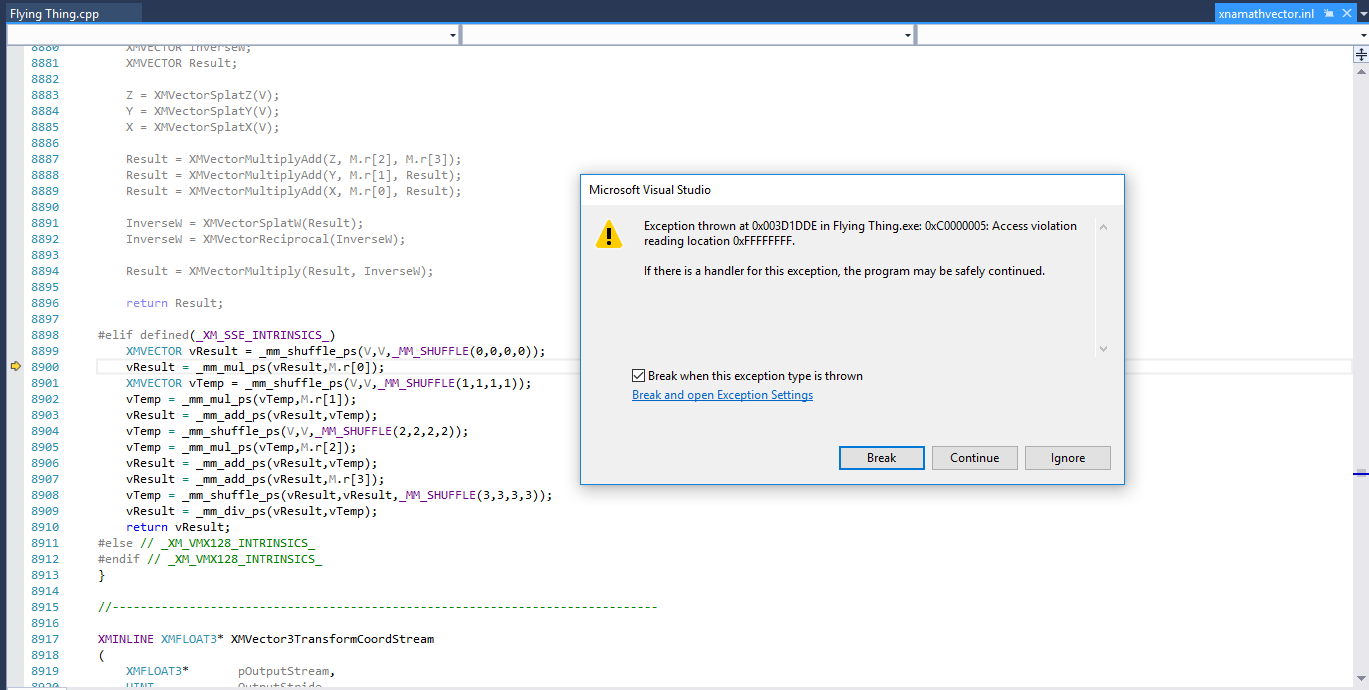
The pig renders well, and the cup appears to look as designed too (although I am unsure what the texture inside the cup is supposed to look like, the “.mtl” file suggests there is different lighting inside at the very least). When rendering the cube it is obvious that some errors are being made. Since there are only 2 triangles per face, when a mistake is made it cannot be overlooked. I feel that if loading a “.obj” file with enough vertices, the prototype is at an acceptable enough level to merge with part 1. However, if I were to implement this, I would not be able to show off the lighting effects implemented in part 1 due to the fact that I haven’t yet implemented what to do with the information extracted from the “.mtl” file. These contain ambient, specular and diffuse lighting values as well as texture and reflection map statements.

At present, I simply extract the values that may be present in 3 “.mtl” files that I have seen and need to do further investigation into any other information may be found inside. I also need to find a relevant place to put the extracted values so that the shaders can apply them to the loaded mesh (something that time constraints have prevented).

Overall, I am very pleased with part 1 of the project though it has a few issues I’d like to remove or improve upon. Part 2 successfully renders “.obj” files and mostly maps the “.jpg” to them, but is rather basic.

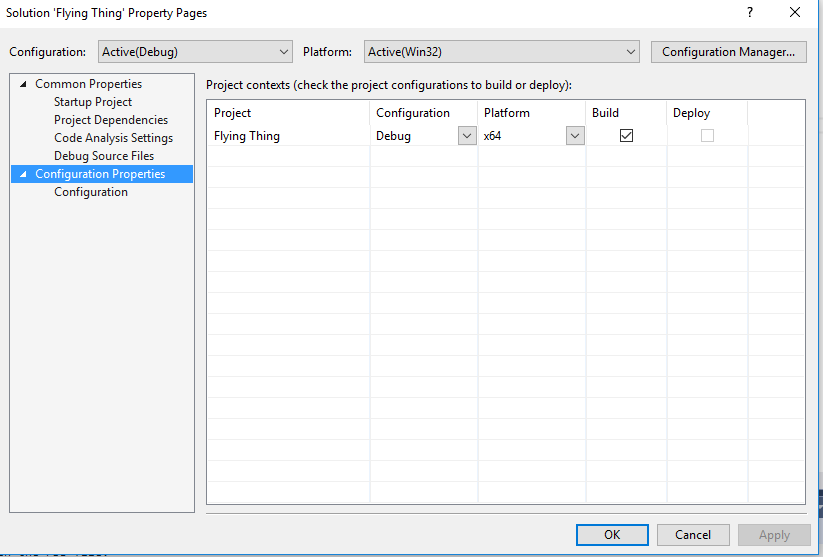
# Appendices

## Appendix 1



Occasional error on start-up of part 1.

## Appendix 2



Potential fix for error on start up on part 1.