Weathering Simulation

Documentation by Doug Kavendek
Advisor: Professor H. Quynh Dinh
Created May 2006; Updated December 2006

Overview

We are trying to develop a weathering model for polygonal meshes through the simulation of particle-model interactions. We will be taking two seemingly unrelated concepts, photon mapping and fluid simulation, to try to provide a pleasing rendition of aging and weathering, which will be represented on the meshes both in textural and geometric changes. Particles in the simulation will be controlled by physical mechanics, reflection, gravitational forces, and an optional vector field for fluid interactions.

Photons

The weathering is performed by the interaction of particles with the models in the program. We will be calling them photons, as some of the motivation behind the project involved using particles in a manner similar to photon mapping. The photons will be projected through the world, affected by various forces, from gravity, initial velocity, or the vector field. Collisions with model faces will result in both geometric perturbations as well as adjustments to residues, both on the faces, and on the particles themselves.

Residue is just a measure of some kind of contaminant on a surface/particle. There are structures for dirt, rust, and moss, but only moss is currently used, to demonstrate how it works. A surface can be initialized to have specific residues, as a photon can be initialized with some residues. Currently, faces start with no residue, and photons start with some moss, and so as photons move around, they deposit some moss. Photons also will pick up residue from the surfaces they encounter. The residue structure also records collision counts for given faces, for testing.

Photons can be fired from the camera, to observe their movement. In this way, there can be a large number of active photons at once. They will move around in the world until they either land on a face or leave the extents. Every time a model-photon collision occurs, there is a chance that the photon may land on the face, rather than reflect, in which case, its residue is deposited and the photon is removed. The amount of geometric distortion from a photon is determined by the velocity of the photon as well as the angle of striking the surface, in addition to a hardness parameter for the surface.

Photons can also be batch-launched, which is far more efficient for performing the weathering. Further controls will be discussed later.

Models

The program takes .PLY models as input. There can be any number of models active at once, and in fact, the main processing constraint is the number of polygons, not the number of models. As a result, a series of low detail models will perform better than a single detailed model.

Currently, models cannot be output after weathering, but this could be implemented fairly easily through freely available libraries for the .PLY file format (or a future group member could write their own routines).

Models are pre-rendered into a display list, which needs to be recompiled whenever there is a change made to the model. Thus, it is far quicker to perform multiple photon simulations before recomputing the model's display list, though it is possible to configure it to refresh on every change, when using the camera photon launcher (see configuration variable `VAR_AUTO_REFRESH`).

Models can be imported into the simulation through the console, by entering "import_model <model file>". Multiple models can be in the simulation at one time, but this can potentially reduce performance somewhat. There is no simple way to remove a model that has been imported, besides restarting the program.

Configuration Variables

There are a series of global configuration variables stored within the CONFIG class, which can be modified several different ways: they can be set with the console (discussed later), or various keys can be bound to affect particular variables (also discussed later). Default values for these variables are set in config.cpp, with the function *init_var()*, which is also used to initialize the variables to contain extra meaning. For example, a variable can be set so that whenever its value changes, the models' display lists are recompiled, along with other settings that are explained in more depth in *CONFIG::init()*.

Pressing F3 will display a list of all the active variables and their current values.

Descriptions

```
VAR_DRAW_FPS            Shows frames-per-second.
VAR_DRAW_AXES           Shows X/Y/Z axes.
VAR_DRAW_EXTENTS_MODE   Cycles through visualization of world extents.
VAR_DRAW_MESSAGES       Shows messages.
VAR_DRAW_VECTORS_MODE   Cycles through visualization of vector field,
                        using the simple method of rendering each
                        vector as a ray.
VAR_DBINS               Shows bins.
VAR_DTRAILS             Show trails on particles.
VAR_SHOW_KEYS           Show key bindings.
VAR_SHOW_VARS           Show variables and values.

VAR_DEBUG_STEP          When set, the simulation will pause after every
                        frame.
```

VAR_FAST_MOVE	When set, the user's camera will move ten times faster.
VAR_FORCE_NEW_LIST	When set to true, the next frame will recompile all display lists.
VAR_FREEZE_PARTICLES	Particles will perform stepping and collision calculations but will not move from velocity (for debugging).
VAR_RUN_SIMULATION	When set, photons will be allowed to step.
VAR_DEBUG_LOG	When set, regular output is directed to err_log.txt (warnings and errors are still output to errors.txt in either case).
VAR_SINGLE_VIEW	When set, only the active viewport is rendered (as opposed to the four-viewport splitscreen).
VAR_CAMERA_LOOK	When set, the user does not need to hold down the camera look key to adjust the view.
VAR_AUTO_REFRESH	When set, any change to a model results in immediate recalculation of display list.
VAR_IN_SIMULATION	While in a batch simulation, this is set.
VAR_PHOTON_ELASTICITY	The maximum elasticity of a photon when it reflects.
VAR_PHOTON_GRAVITY	The bias of gravity towards photons.
VAR_PHOTON_VF_BIAS	The bias of the vector field towards photons.
VAR_PHOTON_CONSERVE	Whether photons conserve momentum or not (velocity is zeroed out for each frame, so there is no acceleration if set to false – this is recommended for use only with positive vector field bias).
VAR_FIRE_STRAIGHT	When photons are fired from the camera and this is set to true, the photons will not be affected by gravity until they collide with a polygon.
VAR_SIM_X0, Y0, etc.	Range of source for batch photon simulation.
VAR_SIM_WAVES	The number of waves of photons to be run when a batch simulation is executed.
VAR_APPLY_ALL	Certain actions can be applied to just the selected model or to all models (e.g. model subdivision) – this variable determines which behavior to use.
VAR_SPREAD_TYPE	Determines how batch photons are fired. Values can be 'ST_RANDOM' (0) – photons start at a random point within the starting field, 'ST_GRID' (1) – photons start regularly spaced within the field, or 'ST_JITTER' (2) – photons start regularly spaced within the field with a slight jitter.
VAR_INCREMENTAL_SUBDIV	When set, subdivision is performed in single steps, rather than subdividing to the max for each wave.
VAR_GIANT_BIN	Must be set before loading models, this will create a bin to encompass the entire model.
VAR_SUBDIV_TARGET	The target size of the smallest polygon to be subdivided.
VAR_SUBDIV_ONLY	Used to create a subdivision-only wave of photons that leave no residue or affect geometry, but rather just determine where subdivisions are needed.

```
VAR_DRAW_VECTORS_MODE2  Better visualization of vector field, using
                        IBFV planes.
VAR_NORMALIZE_IBFV      Determines if IBFV rendering uses normalized
                        vectors.
```

Bins

Particles moving through the world collide with polygons. However, to make collision detection feasible, there is a binning system that divides the world based on position, so that a particle needs to only check its current bin and neighboring bins to see all the polygons that could potentially cause a collision. As polygons are moved and deformed, the bins in which they reside can change.

A limitation of the current scheme is that a polygon's bin is determined entirely by the position of its first vertex. Therefore, if the polygon's face is significantly wider than the size of the bins, it is possible that it won't be detected correctly by particles. This is generally not a problem when you have fairly consistently-sized polygons in a model, but starts to be a problem when you've got multiple models of different sizes in the same world. This problem could be overcome with a different scheme (perhaps one with hierarchical bins), but with the current system there are still workarounds. A model with significantly larger faces can be selected and subdivided until it more closely fits the bin size. This must be done with care, however, because as the polygons get progressively smaller than the bins, the benefits of the binning system are reduced. There seems to be the most efficiency when polygons are somewhere between $1/4^{th}$ and $1/10^{th}$ of a bin's width. Of course, when simulations are run and subdivisions occur, bins are automatically reduced in size slightly, to attempt to keep things running more smoothly.

To effectively disable binning, thereby sacrificing speed for complete accuracy, set the `VAR_GIANT_BIN` variable to true before loading any models. Then, upon loading a model, the bin system is made to encompass the entire model such that every polygon is either in the same bin or a directly adjacent bin (which has the equivalent effect). Using this with a high polygon model is effectively impossible, as simulation time will increase past the order of days, but with special cases may be appropriate.

Controls

The camera is controlled with the mouse. When camera look is enabled (see Configuration Variables), or when you are holding down the spacebar, moving the mouse will rotate the camera. By holding down different combinations of the left and right mouse buttons, the camera can be moved along different planes relative to the camera's look direction. When fast move is enabled, the camera will move significantly faster, which is useful for larger models.

There are many keys mapped to actions, which are defined in control.cpp. Pressing F1 will show all the current keys and a short description of what they do.

Descriptions

ACTION_NULL	Do nothing.
DO_QUIT	Exit and cleanup.
TOGGLE_CONSOLE	Bring down the console, for entering commands.
TOGGLE_DPOLY	Toggle drawing of polygon faces.
TOGGLE_DWIRE_MODE	Toggle drawing of wireframe edges.
TOGGLE_DNORMAL	Toggle drawing of face normals.
TOGGLE_FAST_MOVE	Toggle camera movement speed.
TOGGLE_HIDDEN_LINE	Revert to wireframe mode, but with hidden lines occluded.
CYCLE_GRID_MODE	Display different grid modes. When in perspective view, these are 3D grids along different axes. When in orthogonal view, it is a single, resizable grid.
TOGGLE_DRAW_AXES	Toggle drawing of axes.
CYCLE_DRAW_EXTENTS_MODE	Cycle through extents visualizations.
TOGGLE_MESSAGES	Toggle drawing of messages.
TOGGLE_DRAW_VECTORS_MODE	Cycle through simple vector visualizations.
TOGGLE_DPOINTS	Toggle drawing of points.
TOGGLE_DBINS	Toggle drawing of bins.
TOGGLE_DTRAILS	Toggle drawing of particle trails.
TOGGLE_LIGHTING	Toggle lighting on models.
TOGGLE_DSPAWN_BOX	Toggle drawing of particle spawn box (particles are created inside it).
CYCLE_VISUALIZATION	Cycle through various polygon visualization modes. The color of faces are modulated based on certain parameters. 0: Default brown with residue shown. 1: Normals directly translated into rgb. 2: Curvature. 3: Offset magnitude (offset from original position due to particle collision). 4: Offset vector. 5: Collision count.
RUN_SIMULATION	Run some number of waves of the simulation.
STEP_PHOTONS_ONCE	Allow a single step of the simulation for all photons.
TOGGLE_RUN_SIMULATION	Allows photons to step freely (apart from running a simulation).
SPAWN_PHOTON	Create a test photon from the camera’s position and orientation.
TOGGLE_FIRE_STRAIGHT	When set, photons will fire straight out of the camera and will not be affected by gravity until a collision.
TOGGLE_AUTO_REFRESH	Toggle whether models will refresh immediately upon collisions.
ZOOM_IN/OUT	Zoom within orthogonal views.
GRID_UP/DOWN	Change grid size in orthogonal views.
MOD_CAMERA	When held down, the mouse rotates the camera.
TOGGLE_CAMERA_LOOK	When toggled, the mouse rotates the camera.
TOGGLE_SHOW_KEYS	Toggle display of key bindings.
TOGGLE_SINGLE_VIEW	Toggle between single view and four viewports.
TOGGLE_SHOW_VARS	Toggle display of configuration variables.
BIN_DOWN	Reduce bin sizes (requires refresh of all models).

```
REFRESH                 Recalculates model parameters, chooses bins,
                        and creates display lists.
FULL_SUBDIVISION        Perform subdivision on all polygons.
FORCE_SUBDIVISION       Perform subdivision on only necessary polygons
                        (i.e., polygons that have been collided with
                        test photons that require subdivision).
INVERT_MODEL            In case a model's faces are inside-out, this
                        will reverse them.
TOGGLE_APPLY_ALL        Toggle whether actions should be applied to all
                        models or only selection.
CYCLE_MODEL_SELECTION   Choose selected model.
TOGGLE_INCREMENTAL_SUBDIV Toggle incremental subdivision.

VECTOR_FIELD_TEST       Distort model vertices from vector field.
TOGGLE_DEBUG_LOG        Toggle whether regular output is sent to
                        err_log.txt.
SCREEN_SHOT             Save the current screen into a .BMP file.

SUBDIVIDE_FIELD         Perform subdivision on the vector field.
TOGGLE_DRAW_VECTORS_MODE2 Cycle through IBFV vector field
                        visualization.
DYE_TEST                Toggle visualization of dye in IBFV vector
                        field visualization.
TOGGLE_NORMALIZE_IBFV   Toggles the use of normalized vectors in the
                        IBFV rendering.  This also refreshes the face
                        data used by the IBFV structure.
```

Viewports

The user can view one or four viewports at once. When in four-viewport mode, there is a single projection viewport and three orthogonal viewports, each of which is directed along a different axis. Certain display settings are set per viewport, such as the rendering of polygons or vertices, so to select which viewport is active, click the mouse somewhere inside that viewport. Once active, any viewport-centric display changes will occur only within that viewport. Hitting F3 will toggle between showing the four viewports, and showing just the viewport that was most recently selected.

Selection

A few actions can be applied to either one model or all models within a scene, depending on the state of the configuration variables. If only a single model is to be affected, then the selection of this model can be cycled through all the available models. When cycling through, the name of the model selected is displayed in a message (there is no other visual cue, unfortunately).

The actions that are affected by a selection are the performance of a full subdivision and the inversion of a model's faces. Other actions can be constrained in this way by modifying how they are handled when triggered in control.cpp.

PLY Additions

PLY files allow for extensions within the file itself. There are two additional parameters that have been added to the models: durability and inversion. The durability parameter is a value between zero and one, and determines how much geometric disturbance a model receives from particles. The invert parameter is either a one or zero, and determines the orientation of the faces – if a model is imported with its faces showing inside-out, flip this parameter.

These parameters are simply added to the PLY file above the vertex and face data. As an example, a simple, two polygon face is described below, with the parameter additions underlined. It is defining it such that the durability will be .55, and invert will be set to 0. If no parameters are specified, the defaults are .75 and 1, respectively.

```
ply
format ascii 1.0
comment Example face
element parameters 1
property float32 durability
property uint8 invert
element vertex 4
property float32 x
property float32 y
property float32 z
element face 2
property list uint8 int32 vertex_indices
end_header
0.55 0
0.00 0.00 0.04
0.00 0.04 0.04
0.04 0.04 0.04
0.04 0.00 0.04
3 0 1 2
3 0 2 3
```

More information about the PLY file format can be found here:
http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/

Subdivision

As a model is eroded, it is necessary to subdivide some of its parts to allow a higher level of detail to show the erosion better. A very simple form of subdivision is used to create additional polygons where they are needed. When a collision occurs, if the polygon in question is larger than a given threshold, specified by `VAR_SUBDIV_TARGET`, it is chosen to be subdivided. To keep the model more consistent, the polygon is not subdivided immediately, but rather all polygons in need of subdivision are subdivided after each wave of particles completes.

There are two modes for polygon subdivision: iterative and non-iterative. When it is iterative, a polygon marked for subdivision is only divided once per wave of particles (that is, it is dissected into four smaller sub-triangles). When it is non-iterative, a given polygon is subdivided down until its sub-triangles (or its eventual descendent sub-triangles) are within the given threshold. This latter method has the advantage of a more even distribution of triangles, as well as more consistent erosion.

Simulations

A simulation can be run such that a large number of photons will be fired at the models in the scene, presenting the user with the resulting models after it has completed. The photons are launched from a field specified with the configuration variables `VAR_SIM_X0, VAR_SIM_Y0, VAR_SIM_Z0, VAR_SIM_X1, VAR_SIM_Y1,` and `VAR_SIM_Z_1`. These define opposing corners of a rectangular field. They can be set manually, but whenever a model is inserted into the simulation, these values are adjusted so that it lies directly above all possible points on all models.

The variable `VAR_SPREAD_TYPE` defines how the photons are released from this field. When it is set to `VAR` (0), each photon's position is selected randomly within the field, and the number of photons fired is hard-coded in at 100 – this is acceptable, because its use is not recommended, in favor of the alternatives. The other types, `ST_RANDOM` (1) and `ST_JITTER` (2), fire photons from a regular grid, with the latter type jittering the position of each, and is the recommended type. For both grid-based types, the number of photons is determined by the size of the field, since the spacing between insertions is regular, and set to every .003 units. This generally works fine, because larger models then get a larger number of photons.

When spawned, photons are dropped straight down. This is based on the assumption that gravity is enabled. In the tradeoff between control and usability, usability won, as providing the user the ability to choose how simulation photons are fired could become tricky, so the most common use is the default. Further work could be done in making this more straightforward.

Photon simulations are run in waves. By default, there will be only one wave, but this is controlled by the variable `VAR_SIM_WAVES`. At the end of each wave, any subdivisions necessary are performed. Increasing the number of waves is good approach to increasing weathering effects, because after subdivision has stabilized after a few waves, increasing the number of waves only results in a linear increase in time.

If `VAR_SUBDIV_ONLY` is set, then one additional wave is run per simulation, in addition to the number specified by `VAR_SIM_WAVES`. This additional wave is run first, and it does not affect geometry or residue – it simply sets subdivision tags where appropriate, so that polygons will be subdivided before geometric changes are considered in the next wave. This can potentially provide more accurate results, but tests have not been able to determine a large enough difference between both modes.

Extending Controls

Additional actions can be created to be bound to keys. This requires creating a new action identifier in control.h, and then initializing it in control.cpp. This occurs in *CONTROL::init()*, where calls to *set_primary()* are made (this sets the primary key – to set a secondary key, you would use *set_secondary()* or *set_both()*). This takes as arguments the identifier, a string explanation of the action, the action type, and whether or not it needs to display a waiting screen when the action is executed. Of course, it also takes the key itself.

The action type can be CT_TOGGLE, CT_INSTANT, or CT_SLOW. CT_TOGGLE implies that the action will only occur when a button is pressed initially, and will not occur again until the button is lifted and then pressed again. CT_INSTANT implies that the action will occur for every frame wherein the button is pressed. CT_SLOW implies that the action will occur on the first press, and then periodically will keep occurring a short time later, as long as the key is held.

Extending Configuration Variables

Additional global configuration variables can be created. A new variable identifier must be created in config.h, and then it can be initialized in config.cpp, inside *CONFIG::init()*, with calls to *init_var()*. Each variable can have certain attributes, such as type and whether it should require a refresh of models after the value is modified. It is given a variable string, which is used to identify it and to change its value from the console (discussed later). Additionally, default values can be defined.

Each variable has both a float and a boolean component, and they are referenced different depending on variable type. A VT_BOOL only references the boolean component, a VT_FLOAT only references the float component, and a VT_BOOL_FLOAT references both (if the boolean is true, it is expressed as the float component, but if the boolean is false, it is expressed as zero). Additionally, the VT_BOOL_RANGE functions similarly to a VT_BOOL_FLOAT, except that it can be cycled through a range (defined with *init_var()*) as well as toggled – this is used primarily for cycling through various modes.

Variables can be referenced with *frame.config.get_<type>()*, and supplying the integer variable identifier as a parameter. Likewise, the variables can be set with the various *CONFIG::set()*, *CONFIG::reset()* (forces a variable back to its default), *CONFIG::toggle()*, and so on. It is most useful to simply combine an extension to the configuration variables with an extension to the controls to have some key toggle or change a variable. Numerous examples of this occurring can be found within control.cpp and config.cpp.

The Console

Pressing the '~' button will bring down the console, from which various actions can be executed and various variables can be modified. Using the up/down arrow keys, the user can scroll through past commands as well as partial matches for currently typed commands. The currently defined commands are fairly intuitive: "quit" exits the

program, “toggle,” “set,” and “reset” are the same counterparts to the configuration variable functions (toggle and reset take just the variable string as a parameter, while set takes first the variable string and then the new value – both integer and float inputs are accepted), and “import_model” and “import_field” are used to import a model or vector field file (the proper extensions must be given, and the files must reside in the “/data” directory). Filenames or variable names are given in a match list, similarly to how partial commands are matched.

Extending the Console

Console command identifiers are defined in console.h. They are then initialized with a given string in console.cpp in *CONSOLE::CONSOLE()*, through *command_list[<COM_ID>].set(“<string>”)*. Commands are then handled inside *CONSOLE::process()*, where they are given a separate case block. Inside the case block, a command is given a list of arguments, inside *task.argument[i]*. The number of arguments is given with *task.num_arguments*.

There are several helper functions to output errors to the user if invalid arguments are supplied. *more_arguments()* takes the given number of arguments and the number of expected arguments and creates an error message telling the user to supply the given number of arguments. *bad_argument()* takes the given argument and creates an error message telling the user that the supplied argument is invalid. A common use of *bad_argument()* is, when using the “set” command, to check the output of one of the configuration variable functions it has to call, and if a false is returned, then the variable string must not exist and so a call of *bad_argument()* should be made.

Vector Fields

Vector fields are read in from a file (see the specification for the .VF file format in the file formats section). They specify a cubic set of vectors. The given parameters in the file define the width, dimensions, and origin of the field, such that the given vectors are interpolated spatially through the field. Subdivisions on this field can be handled on initialization through a parameter, or incrementally during execution (through the `SUBDIVIDE_FIELD` command).

When a field is subdivided, its dimensions are effectively doubled, with each new point calculated using some given three dimensional masks. This is more complicated than just interpolating the vectors, and it yield very satisfying results. Inserting just a few vectors can, after subdivision, yield a very complex field – creating a 3x3x3 field with only four non-zero vectors, a nice vortex can be visible after subdividing.

The vector field can either be visualized directly, or through IBFV. Directly, it renders every vector in the field as a single ray. The color of this ray depends on the orientation, such that flows become more evident. However, this can be hard to really discern with just vectors, and also becomes very difficult to look at when a field is dense, so the Image-Based Flow Visualization, or IBFV, is superior.

The IBFV was developed by Jarke J. van Wijk, and a heavily modified version of his code is included. Originally, it was designed to only represent a two dimensional field, so it has been reworked to allow visualization of multiple slices of the field. Each slice is orthogonally aligned with one of three planes, and can be cycled between single slices or all three at once. More slices are possible, but with just three present, a noticeable drop in frame rate occurs, and other methods of visualization (such as stacking a series of slices) do not look visually appealing.

The goal of this field is to modify the behavior of the particles within it. The impact that the vector field has on particles is defined by the variable `VAR_PHOTON_VF_BIAS`. A low bias could simulate wind, while a high bias could simulate water. If `VAR_PHOTON_CONSERVE` is set to false, then photons do not have any acceleration, and so their velocity is set explicitly by the field, which can sometimes be a more desirable effect. Obviously, it is preferred to use a higher density field when not conserving photon velocity, because otherwise the photon movement may appear unnaturally blocky – not that this would necessarily have a large impact on the simulation. Plus, subdividing a few times can solve this.

The vector field structure can handle non-uniform fields, because each element has a position. However, the IBFV, as well as the particle interactions with the field rely on it being uniform, and the current input methods do not allow positions to be specified. Previous versions have been able to import such files, and the functions to do so (such as *build_from_file()* and *build_from_matlab()*) are still present, but commented out of the code.

A vector field can be initialized in the code itself, or can be imported while running. There can be only one vector field active at a time, so subsequent calls to "import_field" from the command console replace the current field with the imported field. Keep in mind that the imported field is created at whatever subdivision level is specified in its file, and may need to be subdivided further if none were specified.

One limitation (or potential feature) of the IBFV rendering method implemented here is that the actual speed of the flow is not represented. Since the units of force have not been standardized, a user could enter a field with vectors of any magnitude. Without accounting for this, if they are too low, the IBFV will not even show movement – it will appear to simply be flickering static. Too high, and the same effect will result, because the offsets will translate pixels too far to indicate any continuity. To counter this, a normalization has been created, which finds the maximum vector magnitude in a field, and scales all the vectors such that that highest vector represents the maximum flow speed for a good representation. By default, vectors are normalized for the visualization (but retain their magnitude for their particle interactions), but this can be toggled, so the IBFV can render it more accurately, using the `TOGGLE_NORMALIZE_IBFV command`.

Files

All files used in this project are listed below, with a brief overview note. For most files, where necessary, there is more particular documentation with the files themselves.

Etc

debug_log.txt	Regular logged output; can be disabled.
errors.txt	Runtime errors and warnings.
Seven.sln, etc.	Visual Studio project files. The name ‘Seven’ is a holdover from an old versioning scheme.

Project files – Files created or modified extensively for this project.

bin.h/cpp	Controls how the space is partitioned. Primarily used for particle/polygon intersection calculations.
FlowSubdiv3D.h/cpp	Subdivision of vector field.
grid.h/cpp	Rendering of grids/axes.
ibfv.h/cpp	Image-Based Flow Visualization. Modified heavily to deal with three dimensions; based off of Jarke J. van Wijk’s original code.
intersection.h/cpp	Efficient ray/polygon intersection calculations, slightly modified from Tomas Möller and Ben Trumbore’s original code.
Mask3D.h/cpp	Three dimensional masks used for vector field subdivision.
model.h/cpp	Everything to do with polygonal models. This reads/parses/creates the models, controls how they are rendered, how particles interact with their polygons, and how their polygons are subdivided.
photon.h/cpp	Handles movement/rendering of photons, which are derived from the PHYS_OBJ class. This also defines the PHOTON_LIST, which contains all active photons and (perhaps confusingly) controls the execution of waves of photons in simulation.
phys_obj.h/cpp	Controls mechanical physics of particles. Movement, acceleration, and collisions (with bounding container or polygons) are handled here.
residue.h/cpp	Specifies how residues are transferred, upon collisions.
rply.h/c	PLY file format helper functions, for reading/writing of PLY files. Write functions are not used.

subdivide.h/cpp	Overview controls of vector field subdivision.
trail.h/cpp	Renders trails for particles.
vectors.h/cpp	Basic vector manipulation functions.
vector_field.h/cpp	Overall control of vector field creation and interaction.
VectorField3D.h/cpp	Structures used for subdividing a vector field

Frame – Files not associated with this project; these are here to provide a basic framework and utilities for the overall program.

bitwise.h/cpp	Simple bitwise operations.
camera.h/cpp	Camera movement/rotation.
config.h/cpp	Controls the visibility and modification of various global variables.
console.h/cpp	Defines the workings of the command-input console.
control.h/cpp	Keyboard mouse input trapped and handled.
define.h	Several symbol definitions.
directory.h/cpp	Directory reading.
display.h/cpp	OpenGL display issues, creating textures, setting up the window, etc.
dynarray.h	Simple functions for dynamically allocating multi-dimensional arrays.
engine.h/cpp	Engine control, mainly in the form of timers.
errlog.h/cpp	Output logging.
extra.h/cpp	Extra display functions.
font.h/cpp	Font building and display of text.
frame.h/cpp	Contains/orders overall execution.
graphic_box.h/cpp	Draws a rectangular box, usually to contain text.
img.h	Simple pixel structure.
key_names.h/cpp	Constructs key binding lists.
list.h/cpp	Templated list class.
magic_begin.h/cpp	Terribly named functions to control some aspects of rendering. Used to efficiently render many objects using the same texture.
main.h/cpp	Start-point of program, and definition of frame and logging objects.
message.h/cpp	Outputs messages to the screen.

overlay.h/cpp	Draws 2D overlay, including the console and messages.
text_box.h/cpp	Draws text overlaid upon a GRAPHIC_BOX. Also, allows a given text box to be active, such that the user can provide input into it. This is only used by the console for this project.
viewport.h/cpp	Handles division of viewports, and viewport-specific variables. Each viewport has its own display variables.

Data files – Files used as input, in the "data" folder. Some are not currently used, and some require further explanation, in a subsequent section of this document.

.BMP	Texture files.
.MSK	Vector field mask files – used in three dimensional subdivision of these fields.
.PLY	Polygonal model files.
x/y_val.txt	Matlab file input for vector fields (not currently used).
.BIN	Binary file input for vector fields (not currently used).
.VF	ASCII file input for vector fields (active format).

File Formats

The PLY file format has already been discussed. BMP is a well-known format, and is simply used for input of textures (mainly used for fonts). The formats for reading Matlab or binary vector field files are not currently in use and have been disabled, but may be used in the future.

The .VF file format is a very simple ASCII file used for input of vector fields, defined by six parameters followed by a series of data points (the exact count of points is determined by the dimensions of the field, provided as a parameter). Each parameter and data point is handled the same – a number surrounded by whitespace, be it spaces, tabs, newlines, etc. The number of data points will always be a multiple of three, because each vector field element has three parts: the X, Y, and Z components.

Parameters – Specified in exactly the order given below.

Dimension	An integer specifying the dimensions of the vector field. Each field is cubic, so only one dimension is needed.
Origin X	The X dimension component of the origin of the vector field. A float.
Origin Y	The Y dimension component of the origin of the vector field. A float.
Origin Z	The Z dimension component of the origin of the vector field. A float.
Width	A floating point value representing the overall width of the entire field.
Subdivisions	Number of times to subdivide a given field on initialization. Useful when specifying a very rough field.

Data Points – A sequential list of data points, containing exactly (dimension^3)*3 distinct numbers. They are specified in X major order, followed by Y, and then Z. For each point, there are three components, as mentioned previously. An example is provided below (comments, including '//' are included here for explanation, but can*not* be present in the data files). Data points are given comments representing which field element in the field's three dimensional vector (in brackets) is being assigned what values (in parenthesis). The use of newlines is purely aesthetic – everything could be on one line, only separated by spaces, or each could be separated by a newline. Dividing each triplet of data points is useful, as each individual vector gets its own line.

```
3                   //Dimension – so here, it is 3x3x3
-.5 -.5 -.5         //Origin, at point (-.5, -.5, -.5)
.5                  //Width – field is .5 units wide.
0                   //Subdivisions – none performed.
0 1 2               // [0,0,0] => (0,1,2)
3 1 0               // [0,0,1] => (3,1,0)
0 2 2               // [0,0,2] => (0,2,2)
0 1 0               // [0,1,0] => (0,1,0)
4 4 4               // [0,1,1] => (4,4,4)
…          //Skipping
1 2 1               // [2,2,1] => (1,2,1)
4 5 0               // [2,2,2] => (4,5,0)
```

User Interface

There are many actions and variables available to the user, through keys and the command console. Many of them are mainly for debugging or very specific use-cases, and will not ever be needed by most users. The more pertinent keys and variables will be discussed next, through simple use-cases, to clarify the basics needed for using this fairly user-unfriendly program. Previously in this document, actions and variables have been referred by their symbolic constant names, but here, their key assignment (for actions) or string representations (for variables) will be used, for easier reference.

On starting the program, there will be no models present, but a vector field will have been loaded. A slice of it should be rendered, along with a representation of the XYZ axes (X is red, Y is green, and Z is blue). To select a different vector field from a file, enter the command "import_field <field filename>" into the console. To bring down the console, hit the '~' key, and then enter your command. To close the console, hit '~' again. To import a model, enter "import_model <model filename>". The file extensions must be included, and if a file entered for either command does not exist, a "bad argument" error appears in the messages.

By importing a new vector field, the previous field is overwritten with the new. However, when importing a new model, the previous model remains, and so multiple models will be present. All models will interact with photons, whether in a simulation, or fired from the camera.

The rendering of models can be modified. Pressing '1' toggles display of polygons, '2' the display of wireframe edges, '3' the display of vertex normals, and '4' the display of vertices. How polygons are colored can be cycled with 'Q'. See the action `CYCLE_VISUALIZATION` in the Controls section for more details.

Pressing F2 will toggle between single-pane and multi-viewport modes. Whichever viewport you have clicked in last is defined as the active viewport, and will be used as the single pane view if F2 is pressed again. Also, changing some display settings in the active viewport may only affect that viewport – for example, one viewport could be rendering the polygons of a model, while another renders only the wireframe. There are two types of grids: 3D and 2D, corresponding to the perspective viewport and orthogonal viewports, respectively. The 3D grid is somewhat confusing, and cycles through different orientations, while the 2D grid can be increased or decreased with the bracket keys ('[' and ']').

The camera is controlled with the mouse. Holding any combination of mouse buttons while moving the mouse will adjust the camera. If "camera_look" is set to 1, then the mouse will always move the camera when the buttons are pressed; otherwise, the spacebar needs to be held down in addition to the mouse buttons. Holding the left mouse button rotates the view, constraining the vertical movement to +/- 90 degrees to keep from getting disoriented. Holding both mouse buttons will allow the camera to be moved forward/backward and left/right along the current looking plane. Holding just the right button will allow the camera to move up/down and left/right along the camera's up vector's plane. It may be strange getting used to this control scheme, but after some time it becomes very natural to move however is desired. Also, the speed at which it moves (but not rotation) as affected by the status of "fast_move" – if set to 1, movement will be ten times faster. This variable can be toggled with the 'M' key.

While in an orthogonal viewport, the mouse will only be able to pan the view around. Zooming is accomplished with the '+' and '-' keys, or with the scroll wheel.

The box from which photons will be spawned during a simulation can be displayed with the 'F' key. This is defined by the "sim_x0", "simy_y0", etc. variables. Pressing 'Y' will run a simulation, given the current settings and number of waves. The number of waves is defined by "sim_waves", and if "subdiv_only" is set to 1, an extra wave of photons is added, purely for subdivision, as mentioned previously. Photons can be fired outside of a simulation, using the 'X' key. They will fire from the camera, in the direction that it is facing.

Regardless of whether a photon is in a simulation or fired from the camera, "photon_fire_straight" controls whether gravity will affect them when they are first launched. That is, if this variable is set to 1, photons will start with a gravity bias of zero, until they collide with a polygon. This is most useful for when firing photons from the camera, because they can be aimed more easily. Photons fired during a simulation will still move downward, but without the acceleration of gravity, because they are given a slight downward velocity. This also has the side-effect of all photons initially hitting the model with the same velocity, which may be more desirable.

To take a BMP screen shot of the current state, press the ',' (the comma). It is useful to disable the rendering of messages first, to make a cleaner shot – this can be done with 'N'. The screen shot function is not perfect, however – in some instances, there is a strange color corruption on some polygons; the cause has not been determined.

The vector field can be rendered two ways – either as a series of vectors, or with the IBFV. The 'V' key cycles through various vector-based rendering methods, while the 'C' key cycles through the IBFV methods. For each, the various methods are divided among displaying everything, single slices, or nothing. The IBFV rendering can be modified to display a dye, toggled with 'I' – this dye leaves trails along each slice, making the flows more easily discernable, especially for static shots (the default rendering without dyes isn't as obvious when it is not animated). A strange, persistent bug concerning IBFVs is that they are not animated while in a multiple-viewport mode, but this does not affect very much. Also, a vector field can be subdivided with the '.' Key (the period). This increases the detail of a field, but it also can take a long time, if a field is already dense – it is certainly *not* a linear function, so be careful with applying it too many times.

'F1' displays all the possible actions and their bound keys, while 'F3' displays all the active configuration variables and their values.

'F4' will reduce the sizes of all the bins in the world, and then will add all models back into these bins. This can sometimes take a while, and is only necessary when you cannot get the polygons to fit nicely within the bin structure. For example, this is useful if the polygons are significantly smaller than the bins, and so the advantages of the bins are lost. The practical inverse of this is 'F6', which subdivides every polygon in a model, which can be useful if some polygons are too large, and not contained wholly within a bin or its neighbors.

The use of 'F6' is governed by the status of the "apply_all" variable . If set to 1, all models will be subdivided. If not, then the current selected model is subdivided. The selected model can be cycled with 'F10', and which one is selected is indicated through a message to the screen with the model's filename. This isn't very user-friendly,

obviously. Also, the inversion of models is controlled by the "apply_all" variable as well – this is done with the 'F8' key, and it reverses the orientation of all the polygons on a model. If you find it is necessary to do this, you should modify the .PLY file with a different value for the "invert" parameter (see PLY Additions section).

Incremental subdivision has been discussed in the Subdivision section, and it can be toggled with 'F11'. Since the recomputation of the display lists for models is costly, it does not occur unless explicitly requested, or after a simulation is run. This can be forced with 'F5', so that changes from firing test photons from the camera can be seen. 'J' can be used to toggle "auto_refresh", which forces a refresh after every change – this is extremely costly, but can be acceptable on very small models or extremely fast computers (note that this only changes how it works outside of a simulation; within a simulation, it is ignored). Finally, if a model has been hit by photons, not only is the mesh moved, but it may also require subdivision. Hitting 'F7' will allow any necessary subdivision to be performed.

Further Plans

If work continues, there are many human interface issues that will require attention, such as better controls over the running of photon simulations and dynamic vector field creation/modification. Also, there is currently no method to store the resulting models, but doing so would be fairly simple – the polygonal mesh would simply be passed to the appropriate PLY writing functions, along with any additional tags, with slight modifications to the PLY format to account for residue.

The binning system has some flaws, but there are workarounds (see the section on binning). However, a hierarchical bin implementation would solve these problems, as well as increase the bin searching efficiency. This would require recreating the bin structure from scratch, while preserving its interactions with models, which could be difficult at this point.

Some texturing, bump-mapping, and shaders could make the output models more interesting. But then the overall purpose of this program would need to be re-evaluated: is it a tool for just weathering models, or is it also for rendering them as well? It seems that storing the models in a consistent format would be a better approach, leaving the rendering for more able applications.

The actual simulation steps require more tweaking and modification. Small changes can sometimes drastically increase the calculation times, so it is hard to really find a good balance, concerning elasticity of particles, velocities, reflectance, and such parameters. And finally, the residue system would require more attention – as it is, it is extremely basic, and only uses the moss residue. Specifying residues on models (such as rust on iron objects) would help complete its implementation.