AIMS@UNM - Team 4 - Source Code
(Also available at https://github.com/carrollcongress/Ice)

```
//  main.cpp
//  Ice
//  This class file will oversee all others

#include <SDL/SDL.h>

#include <iostream>

#include "IceConfig.h"
#include "Provehamus.hpp"

void theVoidDrawsNear();

int main(int argc, char *argv[])
{
    // insert code here...
    std::cout << "Hello, World!\n";

//  Do OpenGL stuff here.
    inceptoProvehamus();

//  Terminate.
    theVoidDrawsNear();

    return 0;
}

void theVoidDrawsNear()
{
    terminoProvehamus();
}

#ifndef PROVEHAMUS_H
#define PROVEHAMUS_H
//
//  Provehamus.h
//  Ice
//

#define GL_GLEXT_PROTOTYPES

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>

#include <SDL/SDL.h>
#include <SDL/SDL_opengl.h>

#include <agar/core.h>
#include <agar/gui.h>
#include <agar/gui/opengl.h>
#include <agar/gui/cursors.h>
#include <agar/gui/sdl.h>
```

```
#include <boost/thread.hpp>
#include <boost/date_time.hpp>
#include <boost/interprocess/ipc/message_queue.hpp>

#include <libxml/parser.h>
#include <libxml/tree.h>

#include "Core.hpp"

#define BUFFER_OFFSET(i) ((char*)NULL + (i))
#define pi 3.14159265358979323846264338327795
#define inc .2
#define rad 2


enum VType {VGrid, VMass, VCharge};

struct VBO
{
    GLuint id;
    size_t size;
    GLfloat *array;
};

//SDL
void inceptoProvehamus();
void terminoProvehamus();
bool vestibulumReactor(int fullscreen);
void setupWindow();

//Agar
static int TrapMouseEvent(const AG_DriverEvent *ev);
static int keyCall(const AG_DriverEvent *ev);
int eventProc(AG_Driver *drv, AG_DriverEvent *ev);
void eventLoop(AG_Driver *drv);
void menuEventHandler(AG_Event *event);

//OpenGL
bool initgl();
void clean_up();

//Core interaction
dataplexor* loadScenario(std::string path);
void loadSnapshot();
void setupScenario(dataplexor *data);
void startScenario();
void stopScenario();

//Tests
void oglTest();
void coreTest();
#endif PROVEHAMUS_H

//
//  Provehamus.cpp
//  Ice
//  Provehamus is the provehamus for this code.
//
```

```
//This part of the code is in Latin as an

#include "Provehamus.hpp"

namespace var
{
int frame_start_time = 0;
int frame_current_time = 0;
int frame_count = 0;
}

SDL_VideoInfo* sInfo;

int showUI = 1;
int bgFocused = 0;
int fps = 60;
bool quit = false, oglt = true, VB0D = false;

enum keypress {UP_KEY, DOWN_KEY, LEFT_KEY, RIGHT_KEY, W_KEY, A_KEY, S_KEY,
D_KEY};
std::map<keypress,bool> kp;

float yrot, xrot;
float xvelocity, zvelocity, zvelocitycache;
vector position, rotation;

float mProjection[16];  /* Projection matrix to load for 'background' */
float mModelview[16];   /* Modelview matrix to load for 'background' */
float mTexture[16];     /* Texture matrix to load for 'background' */

static SDL_Surface *screen;

boost::thread *fusion;

std::map<VType,VBO> VBOMap;

static void menuQuit(AG_Event *event)
{
     AG_Quit();
}
//Scenario Control
void xmlIterate(xmlNode *node, dataplexor *data)
{
}
static void newScenario(AG_Event *event)
{
     //AG_FileDlg *fd = (AG_FileDlg*)AG_SELF();
     char *file = AG_STRING(1);
     //AG_FileType *ft = (AG_FileType*)AG_PTR(2);

     dataplexor *data = new dataplexor;
     xmlDocPtr doc;
    doc = xmlReadFile(file, NULL, 0);
    if (doc == NULL) {
        fprintf(stderr, "Failed to parse %s\n", file);
            return;
    }
```

```
	xmlNode *root = xmlDocGetRootElement(doc);

	xmlNode *cur_node = NULL, *child = NULL, *child2 = NULL;
	xmlAttr *attr = NULL;

	int xA,yA,zA;
	//Load Scenario Properties

	for(attr = root->properties; attr; attr = attr->next)
		{
		if(attr->type = XML_ATTRIBUTE_NODE)
			{
			if (std::string((const char*)attr->name) == std::string
("name")) {
				data->simulationName = std::string((const
char*)attr->children->content);
			}
		if (std::string((const char*)attr->name) == std::string("X")) {
				xA = atoi((const char*)attr->children-
>content);
		}
		if (std::string((const char*)attr->name) == std::string("Y")) {
				yA = atoi((const char*)attr->children-
>content);
		}
		if (std::string((const char*)attr->name) == std::string("Z")) {
				zA = atoi((const char*)attr->children-
>content);
		}
		if (std::string((const char*)attr->name) == std::string
("length")) {
				data->cubeSize = atoi((const char*)attr-
>children->content);
		}
		}
		}
	int size;
	if(xA > zA)
		size = xA;
	else
		size = zA;
	if(yA > size)
		size = yA;

	glGenBuffers(1, &VBOMap[VGrid].id);
	glGenBuffers(1, &VBOMap[VMass].id);

	VBOMap[VGrid].array = (GLfloat*)calloc(pow(size,2)*18, sizeof(GLfloat));
	int u = 0;
	for(int i = 0; i < size; i++)
		{
		for(int x = 0; x < size; x++)
			{
			VBOMap[VGrid].array[u] = x;
			VBOMap[VGrid].array[u+1] = i;
			VBOMap[VGrid].array[u+2] = 0;

			VBOMap[VGrid].array[u+3] = x;
```

```
                        VBOMap[VGrid].array[u+4] = i;
                        VBOMap[VGrid].array[u+5] = size;
                        u+=6;
                        }
                for(int y = 0; y < size; y++)
                        {
                        VBOMap[VGrid].array[u] = 0;
                        VBOMap[VGrid].array[u+1] = y;
                        VBOMap[VGrid].array[u+2] = i;

                        VBOMap[VGrid].array[u+3] = size;
                        VBOMap[VGrid].array[u+4] = y;
                        VBOMap[VGrid].array[u+5] = i;
                        u+=6;
                        }
                for(int z = 0; z < size; z++)
                        {
                        VBOMap[VGrid].array[u] = i;
                        VBOMap[VGrid].array[u+1] = 0;
                        VBOMap[VGrid].array[u+2] = z;

                        VBOMap[VGrid].array[u+3] = i;
                        VBOMap[VGrid].array[u+4] = size;
                        VBOMap[VGrid].array[u+5] = z;
                        u+=6;
                        }
                }
        VBOMap[VGrid].size = u - 5;

        glBindBuffer(GL_ARRAY_BUFFER,VBOMap[VGrid].id);
        glBufferData(GL_ARRAY_BUFFER,sizeof(GLfloat)*VBOMap
[VGrid].size,NULL,GL_STATIC_DRAW);
        glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(GLfloat)*VBOMap
[VGrid].size,VBOMap[VGrid].array);
        glVertexPointer(3,GL_FLOAT,0,0);
        glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);
        glEnableVertexAttribArray(0);


        for (cur_node = root->children; cur_node; cur_node = cur_node->next) {
            if (cur_node->type == XML_ELEMENT_NODE) {
                        std::cout<<cur_node->name<<std::endl;
                        if (strcmp((const char*)cur_node->name,"mass") == 0) {
                              dust mass;
                              int id;
                              for(attr = cur_node->properties; attr; attr = attr-
>next)
                                    {
                                    if (std::string((const char*)attr->name) ==
std::string("name")) {
                                          mass.name = std::string((const char*)
attr->children->content);
                                    }
                                    if (std::string((const char*)attr->name) ==
std::string("mass")) {
                                          mass.mass = atoi((const char*)attr-
>children->content);
                                    }
```

```
                                    if (std::string((const char*)attr->name) ==
std::string("charge")) {
                                          mass.charge = atoi((const char*)attr-
>children->content);
                                    }
                                    if (std::string((const char*)attr->name) ==
std::string("id")) {
                                          id = atoi((const char*)attr->children-
>content);
                                    }
                                    }
                              data->dustArray[id] = mass;
                        }
                        if (strcmp((const char*)cur_node->name,"cube") == 0) {
                              vector i;
                              for(attr = cur_node->properties; attr; attr = attr-
>next)
                                    {
                                    std::cout<<attr->name<<std::endl;
                                    if (std::string((const char*)attr->name) ==
std::string("X")) {
                                          i.x = atoi((const char*)attr->children-
>content);
                                    }
                                    if (std::string((const char*)attr->name) ==
std::string("Y")) {
                                          i.y = atoi((const char*)attr->children-
>content);
                                    }
                                    if (std::string((const char*)attr->name) ==
std::string("Z")) {
                                          i.z = atoi((const char*)attr->children-
>content);
                                    }
                                    }
                              cube *cOBJ = data->cubeAt(i);
                              //cOBJ = new cube();
                              for (child2 = cur_node->children; child2; child2 =
child2->next)
                                    {
                                    if (child2->type == XML_ELEMENT_NODE) {
                                          if (strcmp((const char*)child2-
>name,"medium") == 0) {
                                                int A, B;
                                                for(attr = child2->properties;
attr; attr = attr->next)
                                                      {
                                                      if (std::string((const char*)
attr->name) == std::string("id")) {
                                                            A = atoi((const char*)
attr->children->content);
                                                      }
                                                      if (std::string((const char*)
attr->name) == std::string("mass")) {
                                                            B = atoi((const char*)
attr->children->content);
                                                      }
                                                      }
```

```
                                        cOBJ->addDust(A,B);
                                  }
                            }
                      }

                }
                if (strcmp((const char*)cur_node->name,"conglomerate") ==
0) {

                }
          }
                //Should only be used for testing...
/*
                for(attr = cur_node->properties; attr; attr = attr->next)
                {
                if(attr->type = XML_ATTRIBUTE_NODE)
                      {
                      printf("    Attribute: %s Value:%s\n", attr-
>name,attr->children->content);
                      }

          for(child = cur_node->children; child; child = child->next)
                {
                      if (child->type == XML_ELEMENT_NODE) {
                            printf("node type: Element, name: %s\n", child-
>name);
                      }
                      for(attr = child->properties; attr; attr = attr-
>next)
                            {
                            if(attr->type = XML_ATTRIBUTE_NODE)
                                  printf("    Attribute: %s Value:%s\n",
attr->name,attr->children->content);
                            }
                      }
                }
 */
}
    xmlFreeDoc(doc);
     setupScenario(data);

     std::cout<<data->cubeArray.size()<<std::endl;

     delete(data);

     return;

}
static void pauseScenario(AG_Event *event)
{
     stopScenario();
}
static void resumeScenario(AG_Event *event)
{
     startScenario();
}
static void closeScenario(AG_Event *event)
{
```

```
}

//Window Control
static void preferences(AG_Event *event)
{
}
//File Control
static void openFile(AG_Event *event)
{
     AG_Window *win = AG_WindowNew(0);
    AG_LabelNewStatic(win, 0, "Open Scenario");

     AG_FileDlg* dialog = AG_FileDlgNew(win,AG_FILEDLG_CLOSEWIN|
AG_FILEDLG_LOAD|AG_FILEDLG_EXPAND);
     AG_FileType *filetype = AG_FileDlgAddType(dialog, "Ice XML", "*.ice",
newScenario, NULL);     AG_WindowShow(win);
}
static void saveFile(AG_Event *event)
{
}


void inceptoProvehamus()
{

     position.x = 0;
     position.y = 0;
     position.z = 0;
     yrot = 0;
     xrot = 0;
     rotation.x = 0;
     rotation.y = 0;
     rotation.z = 0;

     zvelocitycache = 0;

    if(!vestibulumReactor(0))
    {

    }
    //Cum finis est per se illato.
//    terminoProvehamus();
}
void terminoProvehamus()
{
    SDL_Quit();
}
bool vestibulumReactor(int plenusscreen)
{
    //SDL_GL_SetAttribute (SDL_GL_DEPTH_SIZE, 24);
    //SDL_GL_SetAttribute (SDL_GL_DOUBLEBUFFER, 1);

    const SDL_VideoInfo *scientiam = SDL_GetVideoInfo();

    Uint32 vexillum = AG_VIDEO_OPENGL;
    int latitudo,altitudo;
    if (plenusscreen)
```

```
{
         vexillum |= AG_VIDEO_FULLSCREEN;
         latitudo = scientiam->current_w;
              altitudo = scientiam->current_h;
}
else
{
    latitudo = 800;
    altitudo = 600;
}
AG_InitCore("Ice", 0);

// Create window
 if (AG_InitVideo(latitudo, altitudo, 32, vexillum) == -1) {
    fprintf(stderr, "%s\n", AG_GetError());
    exit(1);
}
 SDL_Init(SDL_INIT_EVERYTHING);
 initgl();

// Agar initialization
 setupWindow();


glGenBuffers(1, &VBOMap[VGrid].id);
glGenBuffers(1, &VBOMap[VMass].id);

int size = 50;
VBOMap[VGrid].array = (GLfloat*)calloc(pow(size,2)*18, sizeof(GLfloat));
 int u = 0;
 for(int i = 0; i < size; i++)
         {
for(int x = 0; x < size; x++)
         {
         VBOMap[VGrid].array[u] = x;
         VBOMap[VGrid].array[u+1] = i;
         VBOMap[VGrid].array[u+2] = 0;

         VBOMap[VGrid].array[u+3] = x;
         VBOMap[VGrid].array[u+4] = i;
         VBOMap[VGrid].array[u+5] = size;
         u+=6;
         }
 for(int y = 0; y < size; y++)
         {
         VBOMap[VGrid].array[u] = 0;
         VBOMap[VGrid].array[u+1] = y;
         VBOMap[VGrid].array[u+2] = i;

         VBOMap[VGrid].array[u+3] = size;
         VBOMap[VGrid].array[u+4] = y;
         VBOMap[VGrid].array[u+5] = i;
         u+=6;
         }
 for(int z = 0; z < size; z++)
         {
         VBOMap[VGrid].array[u] = i;
         VBOMap[VGrid].array[u+1] = 0;
```

```
            VBOMap[VGrid].array[u+2] = z;

            VBOMap[VGrid].array[u+3] = i;
            VBOMap[VGrid].array[u+4] = size;
            VBOMap[VGrid].array[u+5] = z;
            u+=6;
            }
     }
    VBOMap[VGrid].size = u - 5;

    glBindBuffer(GL_ARRAY_BUFFER,VBOMap[VGrid].id);
    glBufferData(GL_ARRAY_BUFFER,sizeof(GLfloat)*VBOMap
[VGrid].size,NULL,GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(GLfloat)*VBOMap
[VGrid].size,VBOMap[VGrid].array);
    glVertexPointer(3,GL_FLOAT,0,0);
    glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);
    glEnableVertexAttribArray(0);

    eventLoop(AGDRIVER(agDriverSw));
}
void setupWindow()
{
    const SDL_VideoInfo *scientiam = SDL_GetVideoInfo();

    AG_Window *win = AG_WindowNew(0);
    AG_LabelNewStatic(win, 0, "Hello, world!");
    AG_WindowShow(win);

    //Create menubar

    AG_Menu *menu = AG_MenuNewGlobal(0);
    AG_MenuItem *item = AG_MenuNode(menu->root, "Ice", NULL);
            AG_MenuAction(item,"About",NULL,NULL,NULL);
            AG_MenuAction(item,"Preferences...",NULL,NULL,NULL);
            AG_MenuAction(item,"Quit",NULL,menuQuit,NULL);
    item = AG_MenuNode(menu->root, "File", NULL);
            AG_MenuAction(item,"New",NULL,NULL,NULL);
            AG_MenuAction(item,"Open",NULL,openFile,NULL);
            AG_MenuAction(item,"Close",NULL,NULL,NULL);
            AG_MenuAction(item,"Save",NULL,saveFile,NULL);
    item = AG_MenuNode(menu->root, "View", NULL);
            AG_MenuAction(item,"HUD 1",NULL,NULL,NULL);
     item = AG_MenuNode(menu->root, "Control", NULL);
            AG_MenuAction(item,"Start",NULL,resumeScenario,NULL);
            AG_MenuAction(item,"Stop",NULL,pauseScenario,NULL);
    item = AG_MenuNode(menu->root, "Settings", NULL);
            AG_MenuAction(item,"Toggle Fullscreen",NULL,NULL,NULL);

}
void SDLKeyInput(SDL_Event *event)
{
    if(event->type == SDL_KEYDOWN)
    {
       switch(event->key.keysym.sym)
        {
        case SDLK_ESCAPE:
            AG_Quit();
```

```
        break;
        case SDLK_w:
            kp[W_KEY] = true;
            zvelocity = inc;
            break;
        case SDLK_s:
            kp[S_KEY] = true;
            zvelocity = inc;
            break;
        case SDLK_a:
            kp[A_KEY] = true;
            xvelocity = -inc;
            break;
        case SDLK_d:
            kp[D_KEY] = true;
            xvelocity = inc;
            break;
        case SDLK_UP:
            kp[UP_KEY] = true;
                    xrot = -rad;
            break;
        case SDLK_DOWN:
            kp[DOWN_KEY] = true;
                    xrot = rad;
            break;
        case SDLK_LEFT:
            kp[LEFT_KEY] = true;
                    yrot = -rad;
            break;
        case SDLK_RIGHT:
            kp[RIGHT_KEY] = true;
                    yrot = rad;
            break;
    default:
        break;

    }
}
 if(event->type == SDL_KEYUP)
 {
        switch(event->key.keysym.sym)
              {
        case SDLK_w:
        kp[W_KEY] = false;
        zvelocity = 0;
        break;
        case SDLK_s:
        kp[S_KEY] = false;
        zvelocity = 0;
        break;
        case SDLK_a:
        kp[A_KEY] = false;
        xvelocity = 0;
        break;
        case SDLK_d:
        kp[D_KEY] = false;
        xvelocity = 0;
        break;
```

```
            case SDLK_UP:
            kp[UP_KEY] = false;
            xrot = 0;
            break;
            case SDLK_DOWN:
            kp[DOWN_KEY] = false;
            xrot = 0;
            break;
            case SDLK_LEFT:
            kp[LEFT_KEY] = false;
            yrot = 0;
            break;
            case SDLK_RIGHT:
            kp[RIGHT_KEY] = false;
            yrot = 0;
            break;
        default:
            break;
                  }
            }
}
static int TrapMouseEvent(const AG_DriverEvent *ev) //For later use with
mouse/mousewheel.
{
    int rv = 0;
    int x,y;

    switch (ev->type)
    {
    case AG_DRIVER_MOUSE_BUTTON_DOWN:
        x = ev->data.button.x;
        y = ev->data.button.y;

        switch (ev->data.button.which)
        {
        case AG_MOUSE_WHEELUP:
            //vz -= 0.1;
            break;
        case AG_MOUSE_WHEELDOWN:
            //vz += 0.1;
            break;
        }
        return (1);
    }
    return (rv);
}

static int keyCall(const AG_DriverEvent *ev)
{
    return (0);
}

int eventProc(AG_Driver *drv, AG_DriverEvent *ev)
{
    int rv = 0;

    int x,y;
```

```
    switch (ev->type)
    {
    case AG_DRIVER_MOUSE_BUTTON_DOWN:
        x = ev->data.button.x;
        y = ev->data.button.y;

        if (showUI && AG_WindowFocusAtPos(AGDRIVER_SW(drv), x,y))
        {
            if (bgFocused)
            {
                bgFocused = 0;
            }
            return AG_ProcessEvent(drv, ev);
        }
        else
        {
            if (bgFocused)
            {
                  AG_ProcessEvent(drv, ev);
                return TrapMouseEvent(ev);
            }
            else
            {
                bgFocused = 1;
            }
        }
        break;
    case AG_DRIVER_VIDEORESIZE:
        rv = AG_ProcessEvent(drv, ev);
        break;
    case AG_DRIVER_KEY_DOWN:
        if (keyCall(ev))
        {
            return (1);
        }
    default:
        if (!bgFocused && showUI)
        {
            rv = AG_ProcessEvent(drv, ev);
        }
        else
        {
            if (TrapMouseEvent(ev) != 1)
            {
                rv = AG_ProcessEvent(drv, ev);
            }
            else
            {
                return (1);
            }
        }
    }

    return (rv);
}

void translate() //Defines movement of perspective of camera
{
```

```
    double x = (cos((rotation.x / 180) * pi)*cos((rotation.y / 180) * pi)),y=
(sin((rotation.x / 180) * pi)), z = (cos((rotation.x / 180) * pi)*sin
((rotation.y / 180) * pi));

    rotation.x += xrot;
    rotation.y += yrot;

    if(kp[S_KEY])
            {
            position.x += zvelocity*x;
            position.y += zvelocity*y;
            position.z += zvelocity*z;
            }
    else
            {
            position.x -= zvelocity*x;
            position.y -= zvelocity*y;
            position.z -= zvelocity*z;
            }

    if(zvelocity == 0)
            {
            gluLookAt(position.x, position.y, position.z, position.x -
zvelocitycache*x, position.y - zvelocitycache*y, position.z -
zvelocitycache*z, 0.0f, 1.0f, 0.0f);
            }
    else
            {
            gluLookAt(position.x, position.y, position.z, position.x -
zvelocity*x, position.y - zvelocity*y, position.z - zvelocity*z, 0.0f, 1.0f,
0.0f);
            zvelocitycache = zvelocity;
            }
}

void eventLoop(AG_Driver *drv)
{
    SDL_Event event;

    uint now = SDL_GetTicks();
    while(quit == false)
    {
        if(VB0D)
        {
        if(SDL_GetTicks() - now > 20000)
            {
            //UPDATE VAO/VBO
            using namespace boost::interprocess;
            shared_memory_object shm(open_only, memname, read_write);
            mapped_region region(shm, read_write);
            dataplexor *data = static_cast<dataplexor*>(region.get_address
());
            scoped_lock<interprocess_upgradable_mutex> lock(data->mutex);

            //HERE DO WORK
            GLfloat *array = NULL;
            int arraySize = 0;
```

```
            std::map<vector,cube>::iterator it;
            for(it = data->cubeArray.begin(); it != data->cubeArray.end(); it
++)
            {
                int points = floor((.1*(*it).second.mass())/data->cubeSize);
                arraySize += 3*points;
                array = (GLfloat*)realloc(array, sizeof(GLfloat)*arraySize);
                for(int i = 0; i < points; i+=3)
                {
                    array[i] = (*it).second.location.x + rand() % 1 - .5;
                    array[i+1] = (*it).second.location.y + rand() % 1 - .5;
                    array[i+2] = (*it).second.location.z + rand() % 1 - .5;
                }
            }

            glBindBuffer(GL_ARRAY_BUFFER,VBOMap[VMass].id);
            glBufferData(GL_ARRAY_BUFFER,sizeof(GLfloat)
*arraySize,NULL,GL_STATIC_DRAW);
            glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(GLfloat)
*arraySize,array);
            glVertexPointer(3,GL_FLOAT,0,0);
            glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);
            glEnableVertexAttribArray(0);

            lock.unlock();
            now = SDL_GetTicks();
            }
        }
        var::frame_start_time = SDL_GetTicks();
        AG_Window *win;
    AG_BeginRendering(drv);

    glMatrixMode(GL_TEXTURE); glPushMatrix(); glLoadMatrixf(mTexture);
    glMatrixMode(GL_PROJECTION); glPushMatrix(); glLoadMatrixf(mProjection);
    glMatrixMode(GL_MODELVIEW); glPushMatrix(); glLoadMatrixf(mModelview);
    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glEnable(GL_CULL_FACE);
    glDisable(GL_CLIP_PLANE0);
    glDisable(GL_CLIP_PLANE1);
    glDisable(GL_CLIP_PLANE2);
    glDisable(GL_CLIP_PLANE3);
    glClearColor(.5, .5, .5, 0);

    /* Draw our background */
    glLoadIdentity();
    glPushAttrib(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);

    glPushMatrix();
    translate();

    glBindBuffer(GL_ARRAY_BUFFER, VBOMap[VGrid].id);
    glEnableClientState(GL_VERTEX_ARRAY);
//  glEnableClientState(GL_COLOR_ARRAY);
//   std::cout<<VBOMap[VGrid].size<<std::endl;
    glDrawArrays(GL_LINES,0,VBOMap[VGrid].size/3);
//  glDisableClientState(GL_COLOR_ARRAY);
    glDisableClientState(GL_VERTEX_ARRAY);
```

```
if(VB0D)
{
//Draw Grid

//Draw Density
glBindBuffer(GL_ARRAY_BUFFER, VBOMap[VMass].id);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS,0,VBOMap[VMass].size);
glDisableClientState(GL_VERTEX_ARRAY);
}

glBegin(GL_QUADS);
// top
glColor3f(1.0f, 0.0f, 0.0f);
glNormal3f(0.0f, 1.0f, 0.0f);
glVertex3f(-0.5f, 0.5f, 0.5f);
glVertex3f(0.5f, 0.5f, 0.5f);
glVertex3f(0.5f, 0.5f, -0.5f);
glVertex3f(-0.5f, 0.5f, -0.5f);

glEnd();

glBegin(GL_QUADS);
// front
glColor3f(0.0f, 1.0f, 0.0f);
glNormal3f(0.0f, 0.0f, 1.0f);
glVertex3f(0.5f, -0.5f, 0.5f);
glVertex3f(0.5f, 0.5f, 0.5f);
glVertex3f(-0.5f, 0.5f, 0.5f);
glVertex3f(-0.5f, -0.5f, 0.5f);

glEnd();

glBegin(GL_QUADS);
// right
glColor3f(0.0f, 0.0f, 1.0f);
glNormal3f(1.0f, 0.0f, 0.0f);
glVertex3f(0.5f, 0.5f, -0.5f);
glVertex3f(0.5f, 0.5f, 0.5f);
glVertex3f(0.5f, -0.5f, 0.5f);
glVertex3f(0.5f, -0.5f, -0.5f);

glEnd();

glBegin(GL_QUADS);
// left
glColor3f(0.0f, 0.0f, 0.5f);
glNormal3f(-1.0f, 0.0f, 0.0f);
glVertex3f(-0.5f, -0.5f, 0.5f);
glVertex3f(-0.5f, 0.5f, 0.5f);
glVertex3f(-0.5f, 0.5f, -0.5f);
glVertex3f(-0.5f, -0.5f, -0.5f);

glEnd();

glBegin(GL_QUADS);
// bottom
```

```
     glColor3f(0.5f, 0.0f, 0.0f);
     glNormal3f(0.0f, -1.0f, 0.0f);
     glVertex3f(0.5f, -0.5f, 0.5f);
     glVertex3f(-0.5f, -0.5f, 0.5f);
     glVertex3f(-0.5f, -0.5f, -0.5f);
     glVertex3f(0.5f, -0.5f, -0.5f);

     glEnd();

     glBegin(GL_QUADS);
     // back
     glColor3f(0.0f, 0.5f, 0.0f);
     glNormal3f(0.0f, 0.0f, -1.0f);
     glVertex3f(0.5f, 0.5f, -0.5f);
     glVertex3f(0.5f, -0.5f, -0.5f);
     glVertex3f(-0.5f, -0.5f, -0.5f);
     glVertex3f(-0.5f, 0.5f, -0.5f);

     glEnd();
     glPopMatrix();

    glMatrixMode(GL_MODELVIEW); glPopMatrix();
    glMatrixMode(GL_TEXTURE); glPopMatrix();
    glMatrixMode(GL_PROJECTION); glPopMatrix();
    glPopAttrib();
         glPopAttrib();
        if (showUI)
        {
            AG_LockVFS(drv);
            AG_FOREACH_WINDOW(win, drv)
            {
                AG_ObjectLock(win);
                AG_WindowDraw(win);
                AG_ObjectUnlock(win);
            }
            AG_UnlockVFS(drv);
        }
        AG_EndRendering(drv);


        var::frame_count++;
        var::frame_current_time = SDL_GetTicks();

        if((var::frame_current_time - var::frame_start_time) < (1000/60))
        {
            var::frame_count = 0;
            SDL_Delay((1000/60) - (var::frame_current_time -
var::frame_start_time));
        }

        if (SDL_PollEvent(&event) != 0)
        {
            do
            {
                if(event.type == SDL_QUIT)
                {
                    quit = true;
                }
```

```
                if(event.type == SDL_KEYDOWN || event.type == SDL_KEYUP)
                {
                    SDLKeyInput(&event);
                }
                AG_DriverEvent ev;
                AG_SDL_TranslateEvent(drv, (const SDL_Event*)&event, &ev);
                if (eventProc(drv, &ev) == -1)
                    return;
            }
            while (SDL_PollEvent(&event) != 0);
        }
    }
}

bool initgl()
{
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
         GLdouble xMin, xMax, yMin, yMax;
    /* Set a 60 degrees field of view with 1.0 aspect ratio. */
    yMax = 0.01*tan(0.523598f);
    yMin = -yMax;
    xMin = yMin;
    xMax = yMax;
    glFrustum(xMin, xMax, yMin, yMax, 0.01, 100.0);

    //glCullFace( GL_BACK );
    //glFrontFace( GL_CCW );
    //glEnable( GL_CULL_FACE );
     /*
    glEnable(GL_TEXTURE_2D );
    glEnable(GL_DEPTH_TEST);
     glEnable(GL_RESCALE_NORMAL);
    glDepthFunc(GL_ALWAYS);
*/

     //glViewport(0, 0, info->current_w, info->current_h);
/*
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();


     GLdouble aspect = sInfo->current_w/sInfo->current_h;
    gluPerspective(100.0f, aspect, 0.1f, 10000.0f);
 */
/*
     GLdouble fW, fH;
     GLdouble fov = 100.0f;
     fH = tan( (fov / 2) / 180 * pi ) * 0.1f;
     fH = tan( fov / 360 * pi ) * 0.1f;
     fW = fH * aspect;
*/
     //glFrustum( -fW, fW, -fH, fH, 0.1f, 50000.0f );
    glGetFloatv(GL_PROJECTION_MATRIX, mProjection);
    glGetFloatv(GL_MODELVIEW_MATRIX, mModelview);
    glGetFloatv(GL_TEXTURE_MATRIX, mTexture);
```

```
    glMatrixMode(GL_PROJECTION); glPopMatrix();
    glMatrixMode(GL_MODELVIEW); glPopMatrix();
    glMatrixMode(GL_TEXTURE); glPopMatrix();

     glClearDepth(1.0f);                         // Depth Buffer Setup
     glEnable(GL_DEPTH_TEST);                        // Enables Depth Testing
     glDepthFunc(GL_LEQUAL);

    //glEnable(GL_LIGHTING);
    //GLfloat global_ambient[] = { 0.75f, 0.75f, 0.75f, 1.0f };
    //glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);

    if(glGetError() != GL_NO_ERROR)
    {
        return false;
    }
    return true;
}

//Test Whatnot

void oglTest()
{

}
void coreTest()
{
    dataplexor *space = new dataplexor();
    cube *object;

    int i = 0;
    for(int u = 0; u < 4; u++)
    {
        for(int y = 0; y < 4; y++)
        {
            for(int t = 0; t < 4; t++)
            {
                object = new cube(vector(u,y,t));
                space->
                cubeArray[vector(u,y,t)] = *object;
                i++;
            }
        }
    }

    setupScenario(space);
}
void setupScenario(dataplexor *data)
{
     //First, we must create the shared memory that the scenario is going to
be stored in.
   using namespace boost::interprocess;
   try{
      shared_memory_object::remove(memname);
      shared_memory_object shm (create_only, memname, read_write);
      shm.truncate(sizeof(*data));
      mapped_region region(shm, read_write);
      memcpy(region.get_address(),data,sizeof(*data));
```

```
      }
   catch(interprocess_exception &ex){
      shared_memory_object::remove(memname);
      std::cout << ex.what() << std::endl;
   }
}
void startScenario()
{
    fusion = new boost::thread(initiateFusion);
    fusion->detach();
}
void stopScenario()
{
    //Send message stating for end of round.
    //Wait for x seconds, if message isn't returned kill fusion thread.
    using namespace boost::interprocess;
    try{
    shared_memory_object shm(open_only, memname, read_write);
   mapped_region region(shm, read_write);
    dataplexor *data = static_cast<dataplexor*>(region.get_address());
    scoped_lock<interprocess_upgradable_mutex> lock(data->mutex);
    data->halt = true;
    lock.unlock();
    }
    catch(interprocess_exception &ex){
           shared_memory_object::remove(memname);
           std::cout << ex.what() << std::endl;
    }
}
void clean_up()
{
}

#ifndef CORE_H
#define CORE_H
//
//  Core.h
//  Ice
//

#include <stdio.h>
#include <stdlib.h>

#include <boost/thread.hpp>
#include <boost/date_time.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/ipc/message_queue.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/sharable_lock.hpp>
#include <boost/interprocess/sync/upgradable_lock.hpp>
#include <boost/interprocess/sync/interprocess_upgradable_mutex.hpp>

//#include "Provehamus.h"
/*

 Our data:
 A. Exists in some coherant format.
```

B. Have a layer of separate abstraction that is separate from the that determines the amount of power dedicated to sections of space.

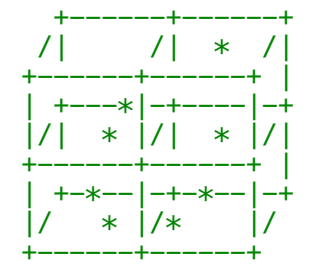

Figure 1.

At its base level, each cube is going to look like Figure 2: A set of cubes that have such and such a quantity of particles, energy, and other such relevant properties. Our code will manipulate each cube's variables based upon surrounding space: a.k.a a standard n-body model at this point. However, n-body models can only go so far when one is dealing with such a large quantity of matter and space. This is where we introduce the statistical model, and incorporate it on a small scale.

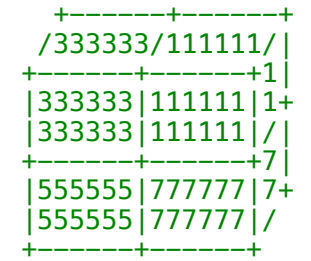

Figure 2.

A statistical model would rely in this situation upon more general descriptions, and can only produce approximate results; it's applied statistics rather than physics. However, it has the bonus of being far less taxing upon the processor, and allows one to work with far larger scenarios.
This code will implement both. At its absolute smallest level, it will not be dealing with particles, but with densities of particles, and other such statistically manipulatable variables. It will resemble Figure 2 far more than Figure 1. One can, however, apply n-body simulation characteristics to this statistical model, through treating sectors of space as particles of cumulative mass, and using this to simplify the calculations.
However, at this point we are left with an n-body simulation with statistical characteristics, one that will likely still consume too much power. Optimization will be a key component here in assuring a runable simulation. There are two
*/

```
#define memname "ice_plexor"
#define memcopyname "ice_plexor_copy"
#define gconst 0.0000000000667384
#define unit 1
#define time 100000//(seconds)

class vector
{
public:
```

```
	vector(){
		x = 0;
		y = 0;
		z = 0;};
	vector(int X, int Y, int Z)
	{
	x = X;
	y = Y;
	z = Z;
	};
	bool operator<(const vector& v) const
	{
		if(x<v.x)
			return -1;
		if(y<v.y)
			return -1;
		if(z<v.z)
			return -1;
		return 1;
	};
	bool operator!=(const vector& v) const
	{
		if(x<v.x)
			return 1;
		if(y<v.y)
			return 1;
		if(z<v.z)
			return 1;
		return -1;
	};
	double x,y,z;
};

//Core Functions
void initiateFusion();
void terminateFusion();

// Stellar Classes
//Dust is used here in place of a better word; it does not literally mean
"dust," but rather a mass of some type.
struct dust
{
	std::string name;
	double mass;
	double charge;
};

class cube {
public:
    cube(){};
	cube(vector loc)
	{
	location = loc;
	};
	vector location;
	//List of pairs, of mediums and their total mass
	std::map<int,double> masses;
```

```
     double mass();
     double charge();

     vector direction;//Force in Newtons

     void addDust(int dustID, double mass);
     void removeDust(int dustID, double mass);
};

class conglomerate: public cube{

};

class dataplexor {
     public:
     //Simulation Et Cetera
     std::string simulationName;
     double cubeSize; //A side's length in meters

     vector size; //A side's length in Cubes
     std::map<vector,cube> cubeArray;
     std::vector<conglomerate> optArray;
     std::map<int,dust> dustArray;

     //Multithreading
     boost::interprocess::interprocess_upgradable_mutex mutex;

     cube* cubeAt(vector loc);
     cube* cubeAt(int x, int y, int z);

    bool halt;
};

class automaton {
    boost::thread m_thread;
     vector location;
public:
     automaton(vector loc);
     ~automaton();
    void start();
    void join();
     bool timedJoin(int seconds);
    void callUponKronos(vector loc);
};
#endif CORE_H

//
//  Core.cpp
//  Ice
//  This class file is the core of the simulation.


#include "Core.hpp"
double cube::mass()
{
     double mass = 0;
     std::map<int,double>::iterator it;
     for(it = masses.begin(); it != masses.end(); it++)
```

```
            {
            std::cout<<(*it).first<<" "<<(*it).second<<std::endl;
            mass+=(*it).second;
            }
    return mass;
}
double cube::charge()
{
    return 0;
}
void cube::addDust(int dustID, double mass)
{
    masses[dustID] += mass;
}
void cube::removeDust(int dustID, double mass)
{
    masses[dustID] -= mass;
}
automaton::automaton(vector loc)
{
    location.x = loc.x;
    location.y = loc.y;
    location.z = loc.z;
}
automaton::~automaton() //For the worker thread later
{
    join();
}
void automaton::start()
{
    m_thread = boost::thread(&automaton::callUponKronos, this, location);
}
void automaton::join()
{
    m_thread.join();
}
bool automaton::timedJoin(int seconds)
{
    using namespace boost::interprocess;
    try{
    return m_thread.timed_join(boost::posix_time::milliseconds(seconds));
    }
    catch(interprocess_exception &ex){
            std::cout << ex.what() << std::endl;
    }
}
void automaton::callUponKronos(vector loc)
//Tells the worker thread to begin its calculations
//This is where the majority of our calculations are.
{
    //sleep(20);

    location.x = loc.x;
    location.y = loc.y;
    location.z = loc.z;

    using namespace boost::interprocess;
```

```
     shared_memory_object shm1(open_only, memname, read_write);
    mapped_region region1(shm1, read_write);
     dataplexor *plex = static_cast<dataplexor*>(region1.get_address());

     sharable_lock<interprocess_upgradable_mutex> *lock = new
sharable_lock<interprocess_upgradable_mutex>(plex->mutex);

     std::map<vector,cube>::iterator it;
     cube *data = &plex->cubeArray[location];
     vector direction(data->direction.x,data->direction.y,data->direction.z);
     double p, dist, sigma, phi;
    for(it = plex->cubeArray.begin(); it != plex->cubeArray.end(); it++)
    {
     if((*it).first != location)
     {
            //Gravitational Simluation
            //The Math is here
            //The Distance Between Points = (x^2 + y^2 + z^2)^.5
            p = plex->cubeSize*pow(pow((*it).second.location.x-data-
>location.x,2)+pow((*it).second.location.z-data->location.z,2)+pow
((*it).second.location.y-data->location.y,2),.5);

            //The Distance Moved
            dist = pow(time,2)*(gconst*(*it).second.mass())/(2*pow(p,2));

            //Determine Sigma Angle
                  sigma = atan(((*it).second.location.y - data->location.y)/
pow(pow((*it).second.location.z - data->location.z,2) + pow
((*it).second.location.x - data->location.x,2),.5));
            //Determine Phi Angle
                  phi = atan(((*it).second.location.z - data->location.z)/
((*it).second.location.x - data->location.x));

            //Vector Addition
                  direction.x += cos(phi)*cos(sigma)*dist;
                  direction.y += sin(sigma)*dist;
                  direction.z += sin(phi)*cos(sigma)*dist;
            }
    }

    shared_memory_object shm(open_only, memcopyname, read_write);
    mapped_region region(shm, read_write);
     dataplexor *cdata = static_cast<dataplexor*>(region.get_address());
     scoped_lock<interprocess_upgradable_mutex> clock(cdata->mutex);

     cube *newCube = cdata->cubeAt(location);
     double *eight = (double*)calloc(8,sizeof(double));

     float temp;

    //0 = Top-Bottom-Left, 4 = Top-Top-Left, 1 = Top-Bottom-Right, 5 = Top-
Top-Right, 2 = Bottom-Bottom-Left, ...

     //Split Along X-Axis
     if(abs(direction.x - floor(direction.x)) < .5)
        {
        temp = abs(direction.x - floor(direction.x));
        eight[0] = .5 - temp;
```

```
    eight[4] = .5 - temp;
    eight[3] = .5 - temp;
    eight[6] = .5 - temp;
    eight[1] = .5 + temp;
    eight[5] = .5 + temp;
    eight[3] = .5 + temp;
    eight[7] = .5 + temp;
    }
else
    {
    temp = abs(floor(direction.x+1) - direction.x);
    eight[0] = .5 + temp;
    eight[4] = .5 + temp;
    eight[3] = .5 + temp;
    eight[6] = .5 + temp;
    eight[1] = .5 - temp;
    eight[5] = .5 - temp;
    eight[3] = .5 - temp;
    eight[7] = .5 - temp;
    }
//Split Along Z-Axis
if(abs(direction.z - floor(direction.z)) < .5)
    {
    temp = abs(direction.z - floor(direction.z));
    eight[0] = eight[0]*(.5 - temp);
    eight[4] = eight[4]*(.5 - temp);
    eight[1] = eight[1]*(.5 - temp);
    eight[5] = eight[5]*(.5 - temp);
    eight[2] = eight[2]*(.5 + temp);
    eight[6] = eight[6]*(.5 + temp);
    eight[3] = eight[3]*(.5 + temp);
    eight[7] = eight[7]*(.5 + temp);
    }
else
    {
    temp = abs(floor(direction.z+1) - direction.z);
    eight[0] = eight[0]*(.5 + temp);
    eight[4] = eight[4]*(.5 + temp);
    eight[1] = eight[1]*(.5 + temp);
    eight[5] = eight[5]*(.5 + temp);
    eight[2] = eight[2]*(.5 - temp);
    eight[6] = eight[6]*(.5 - temp);
    eight[3] = eight[3]*(.5 - temp);
    eight[7] = eight[7]*(.5 - temp);
    }
//Split Along Y-Axis
if(abs(direction.y - floor(direction.y)) < .5)
    {
    temp = abs(direction.y - floor(direction.y));
    eight[4] = eight[4]*(.5 + temp);
    eight[4] = eight[5]*(.5 + temp);
    eight[6] = eight[6]*(.5 + temp);
    eight[7] = eight[7]*(.5 + temp);
    eight[0] = eight[0]*(.5 - temp);
    eight[1] = eight[1]*(.5 - temp);
    eight[2] = eight[2]*(.5 - temp);
    eight[3] = eight[3]*(.5 - temp);
    }
```

```
        else
        {
        temp = abs(floor(direction.y+1) - direction.y);
        eight[4] = eight[4]*(.5 - temp);
        eight[5] = eight[5]*(.5 - temp);
        eight[6] = eight[6]*(.5 - temp);
        eight[7] = eight[7]*(.5 - temp);
        eight[0] = eight[0]*(.5 + temp);
        eight[1] = eight[1]*(.5 + temp);
        eight[2] = eight[2]*(.5 + temp);
        eight[3] = eight[3]*(.5 + temp);
        }
    //Here we divide mass among the cubes...

    //vector(floor(direction.x-.5)+.5),(floor(direction.y-.5)+.5),(floor
(direction.z+.5)+.5));
    std::map<int,double>::iterator it1;
    for(it1 = data->masses.begin(); it1 != data->masses.end(); it1++)
           {
           cdata->cubeAt((floor(direction.x-.5)+.5),(floor(direction.y-.5)+.
5),(floor(direction.z+.5)+.5))->addDust((*it1).first,eight[0]*
(*it1).second);//Cube 1
    cdata->cubeAt((floor(direction.x+.5)+.5),(floor(direction.y-.5)+.5),
(floor(direction.z+.5)+.5))->addDust((*it1).first,eight[1]*(*it1).second);//
Cube 2
    cdata->cubeAt((floor(direction.x-.5)+.5),(floor(direction.y-.5)+.5),
(floor(direction.z-.5)+.5))->addDust((*it1).first,eight[2]*(*it1).second);//
Cube 3
    cdata->cubeAt((floor(direction.x+.5)+.5),(floor(direction.y-.5)+.5),
(floor(direction.z-.5)+.5))->addDust((*it1).first,eight[3]*(*it1).second);//
Cube 4
    cdata->cubeAt((floor(direction.x-.5)+.5),(floor(direction.y+.5)+.5),
(floor(direction.z+.5)+.5))->addDust((*it1).first,eight[4]*(*it1).second);//
Cube 5
    cdata->cubeAt((floor(direction.x+.5)+.5),(floor(direction.y+.5)+.5),
(floor(direction.z+.5)+.5))->addDust((*it1).first,eight[5]*(*it1).second);//
Cube 6
    cdata->cubeAt((floor(direction.x-.5)+.5),(floor(direction.y+.5)+.5),
(floor(direction.z-.5)+.5))->addDust((*it1).first,eight[6]*(*it1).second);//
Cube 7
    cdata->cubeAt((floor(direction.x+.5)+.5),(floor(direction.y+.5)+.5),
(floor(direction.z-.5)+.5))->addDust((*it1).first,eight[7]*(*it1).second);//
Cube 8
           }
    //     newCube
    //data
    clock.unlock();
    lock->unlock();
}
cube* dataplexor::cubeAt(vector loc)
{
    //if (cubeArray.count(loc) == 0)
           //cubeArray[loc] = cube();
    return &cubeArray[loc];
}
cube* dataplexor::cubeAt(int x, int y, int z)
{
    return &cubeArray[vector(x,y,z)];
```

```
}
void initiateFusion()
{
   //Wait for Scenario to be loaded
    bool stop = false;
    while(stop != true)
    {
     using namespace boost::interprocess;
     try{
            //Read & Lock Scenario
    shared_memory_object shm(open_only, memname, read_write);
    mapped_region region(shm, read_write);
     dataplexor *data = static_cast<dataplexor*>(region.get_address());
     sharable_lock<interprocess_upgradable_mutex> lock(data->mutex);

     //Make Copy
     shared_memory_object::remove(memcopyname);
     shared_memory_object shmB(create_only, memcopyname, read_write);
     shmB.truncate(sizeof(*data));
     mapped_region regionB(shmB, read_write);
     dataplexor *copy = static_cast<dataplexor*>(regionB.get_address());
     memcpy(copy,data,sizeof(*data));

     //Clear Copy Data
     std::map<vector,cube>::iterator it;
     for (it = copy->cubeArray.begin(); it != copy->cubeArray.begin(); it++) {
            (*it).second.masses.clear();
     }

     //Determine the number of physical threads available
            int nthreads;
            if(boost::thread::hardware_concurrency() != 0)
                  nthreads = boost::thread::hardware_concurrency();
            else
                  nthreads = 2;

            std::map<int, bool> completion;
            for (int u = 0; u < nthreads; u++)
                  completion[u] = false;
            int complete = 0;

            //
            std::map<vector,cube>::const_iterator it2;
            std::map<int,automaton*> threads;

            int i = 0;
            for(it2 = data->cubeArray.begin(); it2 != data->cubeArray.begin
(); it2++)
                  {
                  if(i < nthreads)
                        {
                        //Thread Check
                        threads[i] = new automaton(it2->first);
                        threads[i]->start();
                        i++;
                        }
                  else
                        it2 = data->cubeArray.begin();
```

```
                  }
            //
            while (complete != nthreads) {
                  for(int     i = 0; i < nthreads; i++)
                        {
                        if (threads[i]->timedJoin(500))
                        {
                              if(it != copy->cubeArray.end() && !completion
[i])
                                    {
                                    delete(threads[i]);
                                    threads[i] = new automaton((*it2).first);
                                    threads[i]->start();
                                    it2++;
                                    }
                              else
                                    {
                                    completion[i] = true;
                                    complete += 1;
                                    }
                        }
                  }
            }
    //Check to see if pause is requested...
     if(data->halt == true)
     {
         stop = true;
        data->halt = false;
     }
     //Unlock Memory
     lock.unlock();
     }
     //Catch Exceptions
     catch(interprocess_exception &ex){
            shared_memory_object::remove(memname);
            std::cout << ex.what() << std::endl;
     }
    }
}

#CMakeLists.txt
cmake_minimum_required (VERSION 2.6)
# The version number.
project (Ice)

set (Ice_VERSION_MAJOR 1)
set (Ice_VERSION_MINOR 0)

set(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS true)
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})

# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
  "${PROJECT_SOURCE_DIR}/IceConfig.h.in"
  "${PROJECT_BINARY_DIR}/src/IceConfig.h"
  )
```

```
Find_Package ( SDL REQUIRED )
if ( NOT SDL_FOUND )
message ( FATAL_ERROR "SDL not found!" )
endif ( NOT SDL_FOUND )

find_package(OpenGL REQUIRED)
find_package(Boost COMPONENTS system thread filesystem REQUIRED)
find_package(libxml2 REQUIRED)

execute_process(COMMAND agar-config --cflags OUTPUT_VARIABLE AGAR_CFLAGS)
execute_process(COMMAND agar-config --libs OUTPUT_VARIABLE AGAR_LFLAGS)

add_definitions(${AGAR_CFLAGS} ${LIBXML2_DEFINITIONS})
set(CMAKE_EXE_LINKER_FLAGS "-L/usr/lib -lag_gui -lag_core -L/usr/lib -
lfreetype -lz -lbz2  -L/usr/lib -lm -L/usr/lib -ljpeg -L/usr/lib -lpng15 -
lpthread -lrt")

if(APPLE)
set(ALL_OSX_SRCS src/osx/SDLMain.m src/osx/SDLMain.h)
set (OSX_HEADERS "/opt/local/include")
set(CMAKE_EXE_LINKER_FLAGS "-L/usr/local/lib -lag_gui -lag_core -L/opt/local/
lib -Wl,-framework,Cocoa -L/opt/local/lib -lfreetype -lz -lbz2 -framework
OpenGL -lm -L/opt/local/lib -ljpeg -L/opt/local/lib -lpng14 -lpthread")

include_directories(
${OSX_HEADERS}
)
endif()

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
include_directories("${PROJECT_BINARY_DIR}" ${SDL_INCLUDE_DIRS} $
{LIBXML2_INCLUDE_DIR})
FILE(GLOB ALL_CPP_SRCS src/*.cpp)
FILE(GLOB ALL_H_SRCS src/*.hpp)

add_executable(Ice ${ALL_CPP_SRCS} ${ALL_H_SRCS} ${ALL_D_SRCS} $
{ALL_OSX_SRCS})

target_link_libraries (Ice
${SDL_LIBRARY}
${GLUT_LIBRARY}
${OPENGL_LIBRARY}
${Boost_FILESYSTEM_LIBRARY}
${Boost_SYSTEM_LIBRARY}
${Boost_THREAD_LIBRARY}
${LIBXML2_LIBRARIES}
)
```