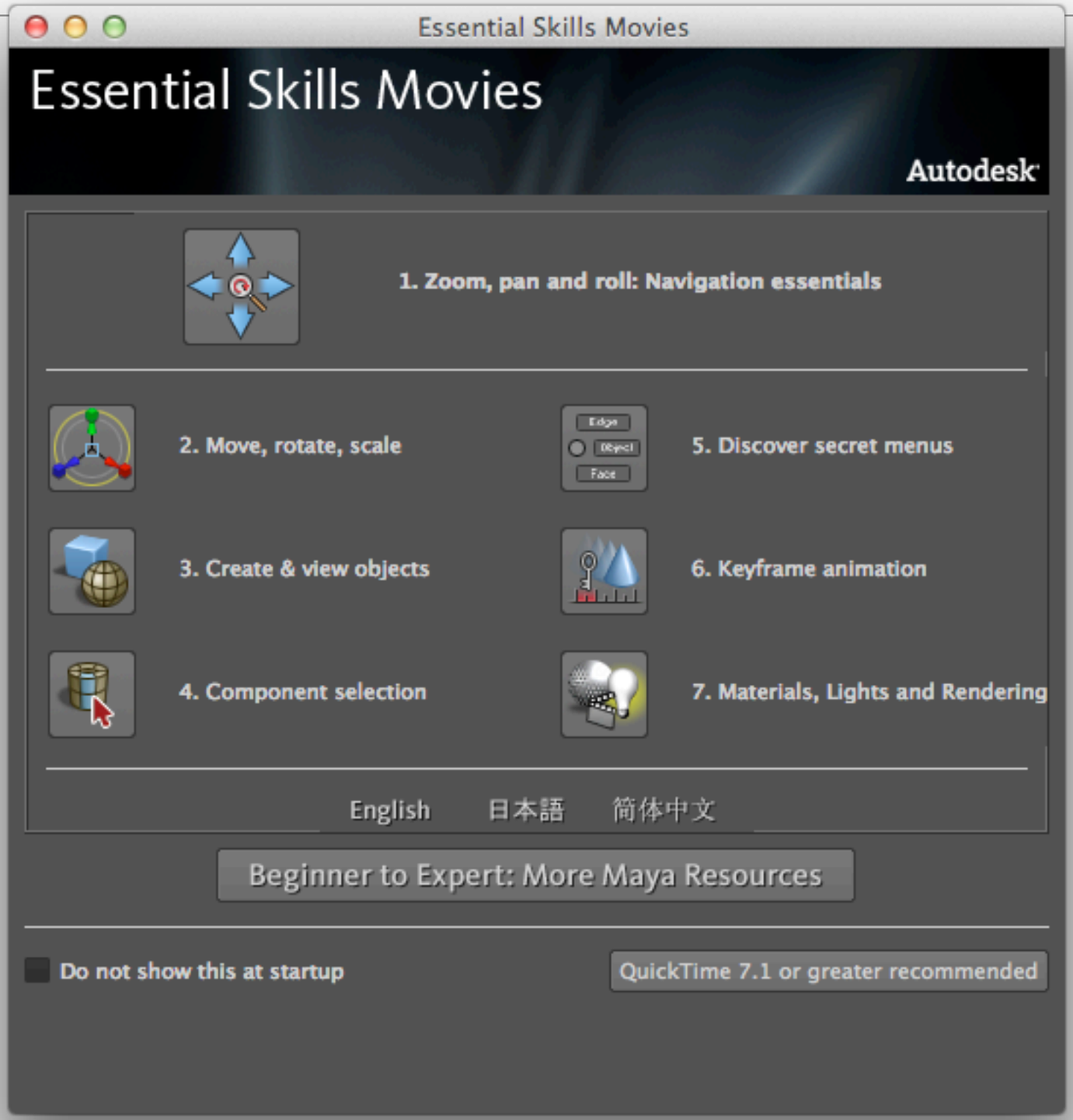
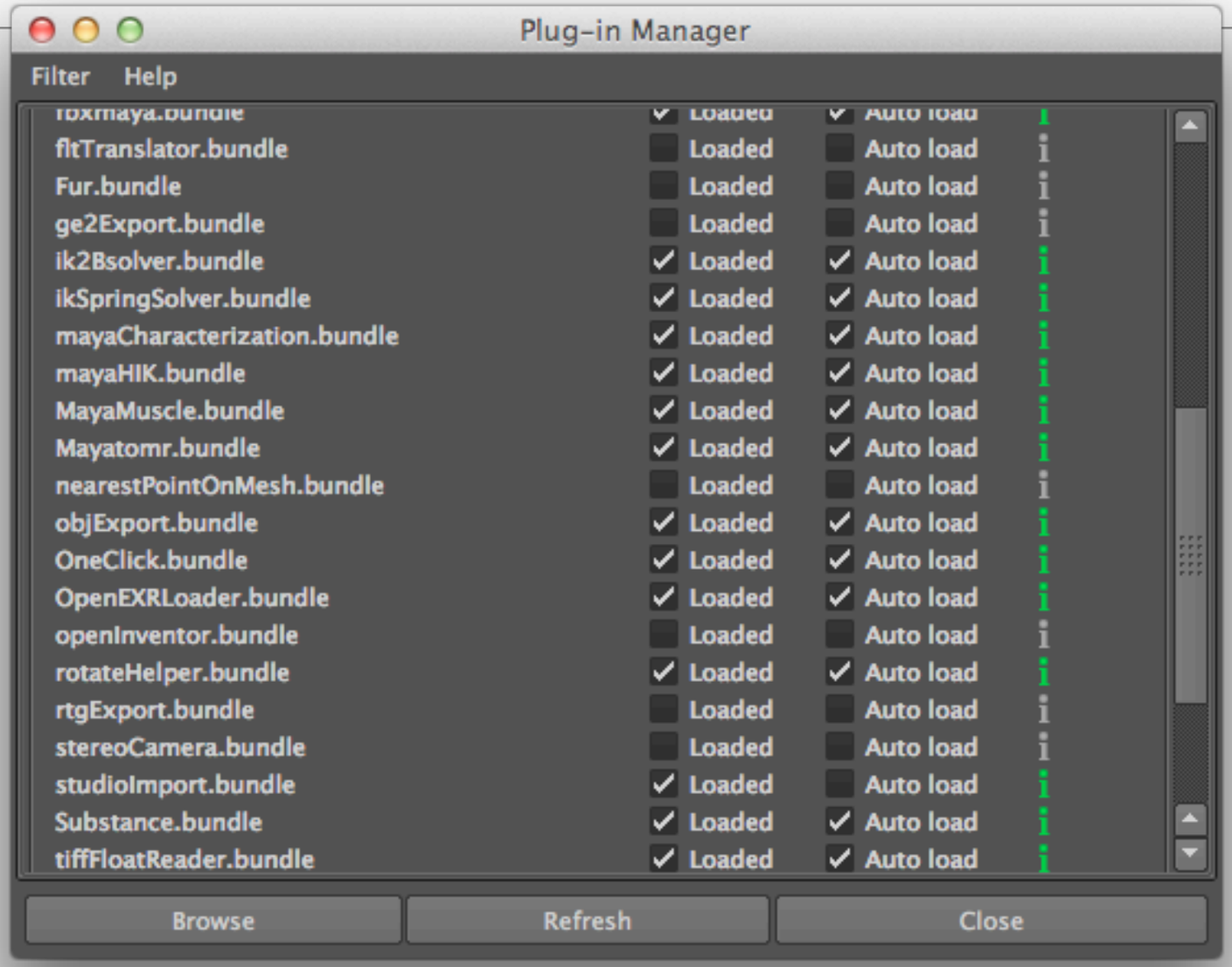
Wavefront

OpenGL

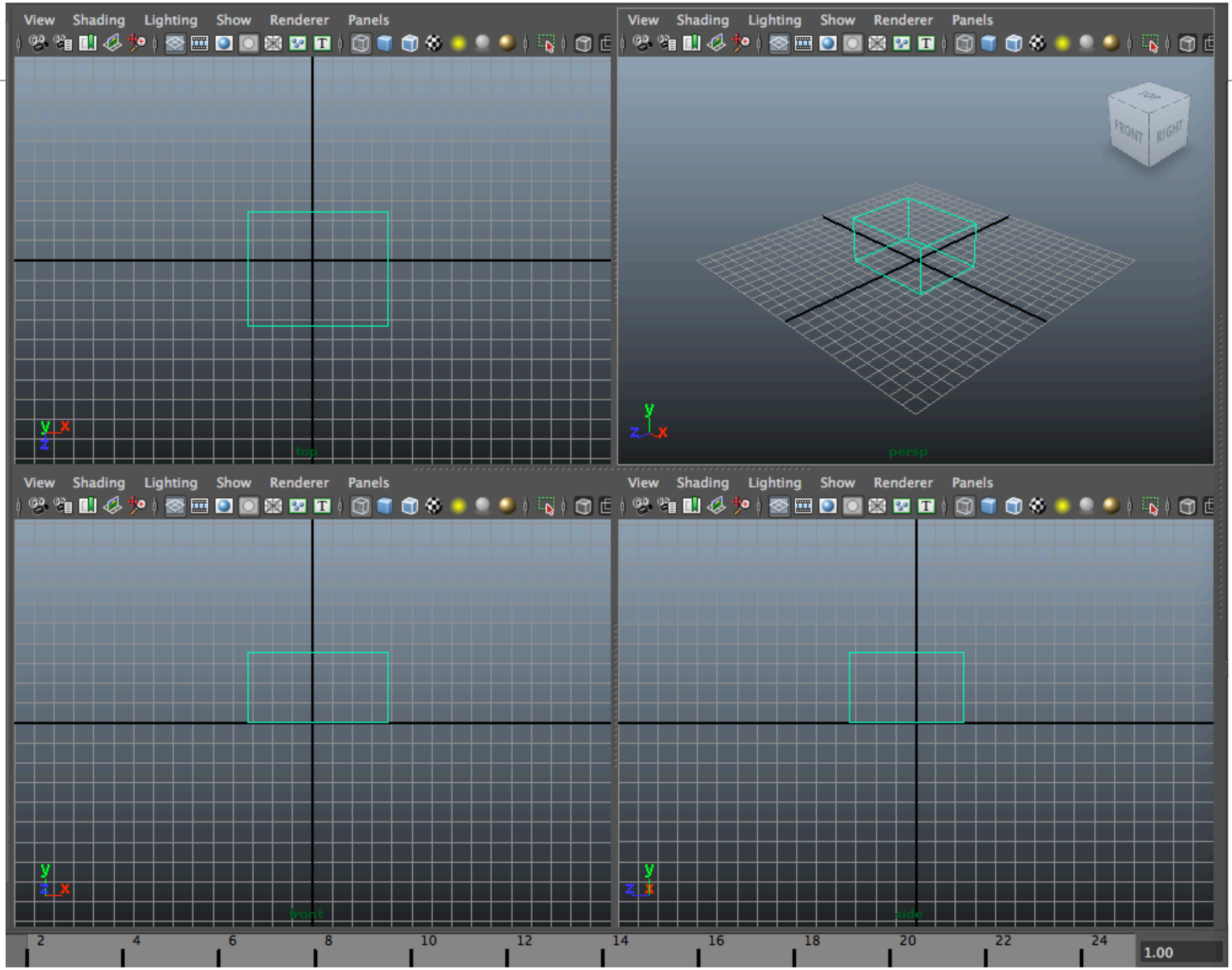
Install Maya 2012 - Student Edition



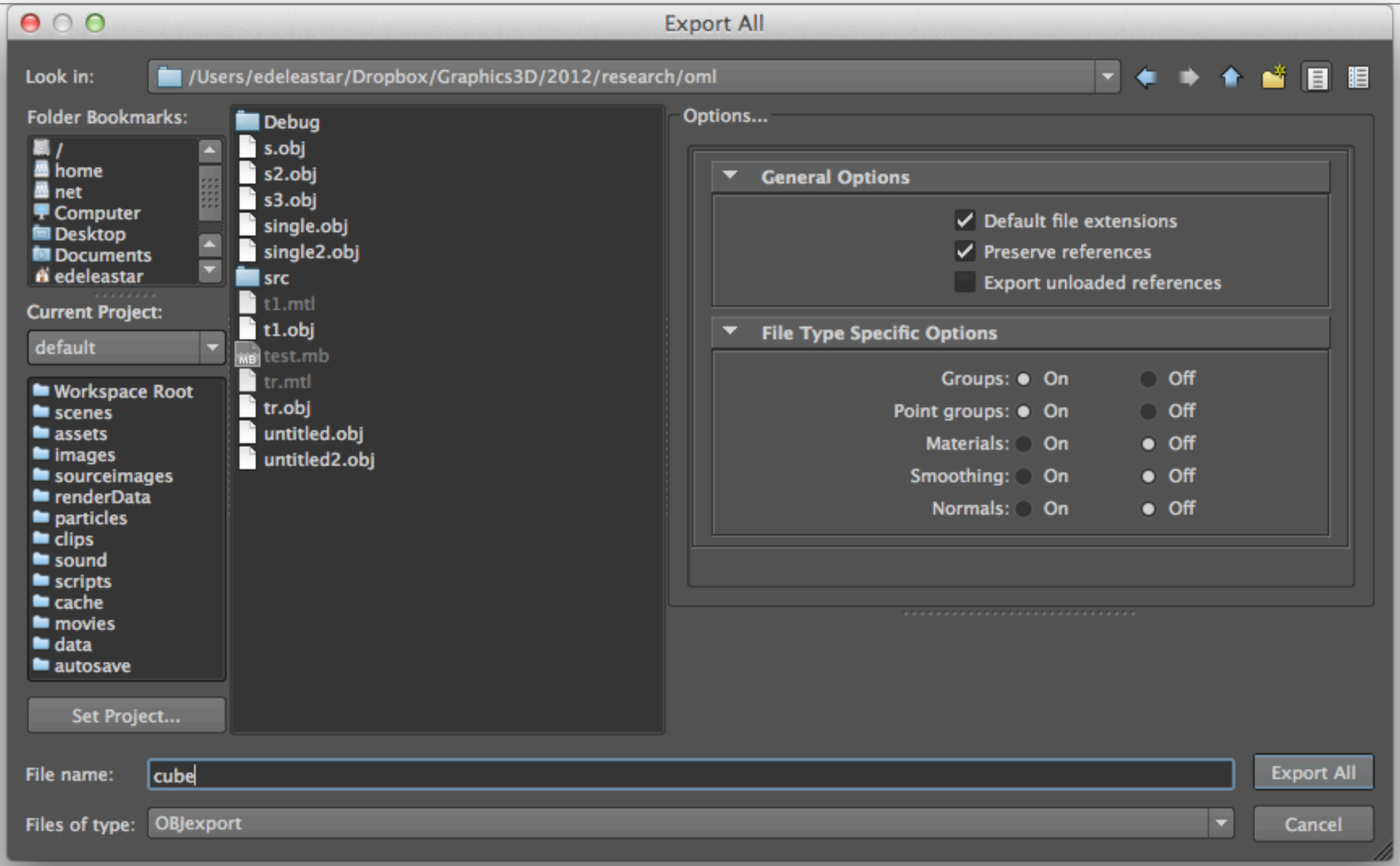
Enable WaveFront plugin



Maya 2012 - Draw a Cube



Export as Wavefront file



cube.obj

```
# This file uses centimeters as units for non-parametric coordinates.

g default
v -3.271605 0.000000 3.333333
v 3.827160 0.000000 3.333333
v -3.271605 3.566200 3.333333
v 3.827160 3.566200 3.333333
v -3.271605 3.566200 -2.407407
v 3.827160 3.566200 -2.407407
v -3.271605 0.000000 -2.407407
v 3.827160 0.000000 -2.407407
vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.625000 0.250000
vt 0.375000 0.500000
vt 0.625000 0.500000
vt 0.375000 0.750000
vt 0.625000 0.750000
vt 0.375000 1.000000
vt 0.625000 1.000000
vt 0.875000 0.000000
vt 0.875000 0.250000
vt 0.125000 0.000000
vt 0.125000 0.250000
g pCube1
f 1/1 2/2 4/4 3/3
f 3/3 4/4 6/6 5/5
f 5/5 6/6 8/8 7/7
f 7/7 8/8 2/10 1/9
f 2/2 8/11 6/12 4/4
f 7/13 1/1 3/3 5/14
```

File Format...

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Français
Português
Русский
Svenska
Українська

Wavefront .obj file

From Wikipedia, the free encyclopedia

For other uses, see Obj (disambiguation).

OBJ (or .OBJ) is a geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors. For the most part it is a universally accepted format.

The OBJ file format is a simple data-format that represents 3D geometry alone — namely, the position of each vertex, the UV position of each texture coordinate vertex, normals, and the faces that make each polygon defined as a list of vertices, and texture vertices. Vertices are stored in a counter-clockwise order by default, making explicit declaration of normals unnecessary.

OBJ geometry format

Filename extension	`.obj`
Internet media type	`application/x-tgif`
Developed by	Wavefront Technologies
Type of format	3D model format

Contents [hide]

World Abstractions

- Color
- Vector3
- World
- Main
- Model
- ModelObject

Color & Vector3

```
struct Color
{
  float R;
  float G;
  float B;
  float A;

  static Color White;
  static Color Yellow;
  static Color Red;
  static Color Magenta;
  static Color Cyan;
  static Color Green;
  static Color Black;
  static Color Blue;


  Color();
  Color(float r, float g, float b, float a=1.0f);
  Color(int r, int g, int b, int a=255);

  void render();
  void renderClear();
};
```

```
struct Vector3
{
  float X;
  float Y;
  float Z;

  static Vector3 UnitX;
  static Vector3 UnitY;
  static Vector3 UnitZ;


  Vector3(float x, float y, float z);
  Vector3(float value);
  Vector3();
  Vector3(std::istream& is);

  void translate();
  void rotate (float angle);

  void render();
};
```

World

```
#define theWorld World::GetInstance()

class World
{
  public:
    static World& GetInstance();

    void setCmdlineParams(int*argc, char **argv);
    void initialize(int width, int height, std::string name);
    void start();
    void loadModel (std::string modelName);

    void render();
    void keyPress(unsigned char ch);

  private:
    static World* s_World;
    Model theModel;
    int   *argc;
    char **argv;
};
```

main

```
int main(int argc, char* argv[])
{
  theWorld.setCmdlineParams(&argc, argv);
  theWorld.initialize(800,600, "First World");

  theWorld.loadModel("cube.obj");
  theWorld.start();
  return 0;
}
```

load & render

```
void World::render()
{
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  // ???
  glutSwapBuffers();
}

void World::loadModel (std::string modelName)
{
  ifstream inStream;
  inStream.open(modelName.c_str(), ios::in);
  if (!inStream.fail())
  {
    //???
  }
}
```

Model

```
typedef std::map <std::string, ModelObject>  ModelMap;
typedef ModelMap::iterator ModelMapIterator;

struct Model
{
  ModelMap  modelObjects;

  Model();
  bool load(std::istream &is);
  void render();
};
```

ModelObject

```
struct ModelObject
{
  std::string name;
  std::vector<Face> faces;
  std::vector <Vector3> vertices;

  ModelObject();
  ModelObject(std::istream&);
  void render(std::vector <Vector3>&);
};
```

Model Implementation (1)

```
Model::Model()
{
}

bool Model::load(istream& is)
{
  string indicator;
  is >> indicator;
  while (!is.eof())
  {
    if (indicator == "#")
    {
      string buf;
      getline(is, buf);
    }
    else if (indicator == "g")
    {
      ModelObject a(is);
      if (modelObjects.find(a.name) == modelObjects.end())
      {
        modelObjects[a.name] = a;
      }
    }
    is >> indicator;
  }

  return true;
}
```

Model Implementation (2)

```
void Model::render()
{
  ModelMapIterator defaultIter = modelObjects.find("default");
  if (defaultIter != modelObjects.end())
  {
    ModelObject defaultObject(defaultIter->second);
    for (ModelMapIterator iter = modelObjects.begin(); iter != modelObjects.end(); iter++)
    {
      iter->second.render(defaultObject.vertices);
    }
  }
}
```

ModelObject Implementation (1)

```
ModelObject::ModelObject()
{}

ModelObject::ModelObject(istream& is)
{
  string indicator;
  is >> name;
  bool stillGroup=true;
  do
  {
    is >> indicator;
    if (indicator == "v")
    {
      vertices.push_back(Vector3(is));
    }
    else if (indicator == "f")
    {
      faces.push_back(Face(is));
    }
    else if (indicator == "g")
    {
      stillGroup = false;
    }
    else
    {
      string buf;
      getline(is, buf);
    }
  } while (stillGroup && !is.eof());
  is.putback(indicator[0]);
}
```

ModelObject Implementation

```
void ModelObject::render(vector <Vector3>&defaultTable)
{
  cout << " rendering " << name << " with " << faces.size() << "faces" << endl;
  for (unsigned int i = 0; i < faces.size(); i++)
  {
    faces[i].render(defaultTable);
  }
}
```

```
struct Face
{
  int vertices[3];
  int textures[3];

  Face(std::istream& is);
  void render(std::vector <Vector3>&);
};
```

```cpp
using namespace std;

Face::Face(istream& is)
{
  char ch1;
  for (int i = 0; i < 4; i++)
  {
    string separator;
    is >> vertices[i];
    is >> ch1;
    is >> textures[i];
  }
}

void Face::render(std::vector <Vector3>&defaultTable)
{
  glBegin(GL_QUADS);
  for (int i=0; i<4; i++)
  {
    glVertex3f( defaultTable[vertices[i] - 1].X,
                defaultTable[vertices[i] - 1].Y,
                defaultTable[vertices[i] - 1].Z );
    cout << defaultTable[vertices[i] - 1].X << " " << defaultTable[vertices[i] - 1].Y << " " << defaultTable[vertices[i] - 1].Z << endl;
  }
  glEnd();
}
```

Model Loading

```
void World::render()
{
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  Vector3(0,0,-10).translate();
  theModel.render();

  glutSwapBuffers();
}

void World::loadModel (std::string modelName)
{
  ifstream inStream;
  inStream.open(modelName.c_str(), ios::in);
  if (!inStream.fail())
  {
    theModel.load(inStream);
  }
}
```