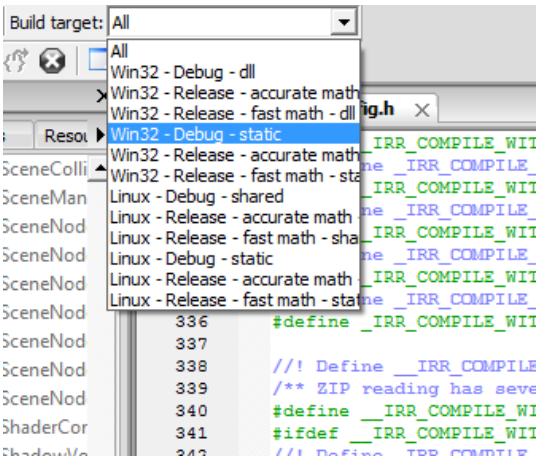
Implementation

First I downloaded the irrlicht engine Software Development Kit and unpacked it, first I will need to build and compile it. Before building I chose to disable some features of the engine I didn't need, to make the program smaller in size. All it took was commenting out or deleting #define macro's in a C++ header file of the engine. First I disabled joysticks as the grass generator will not need that.

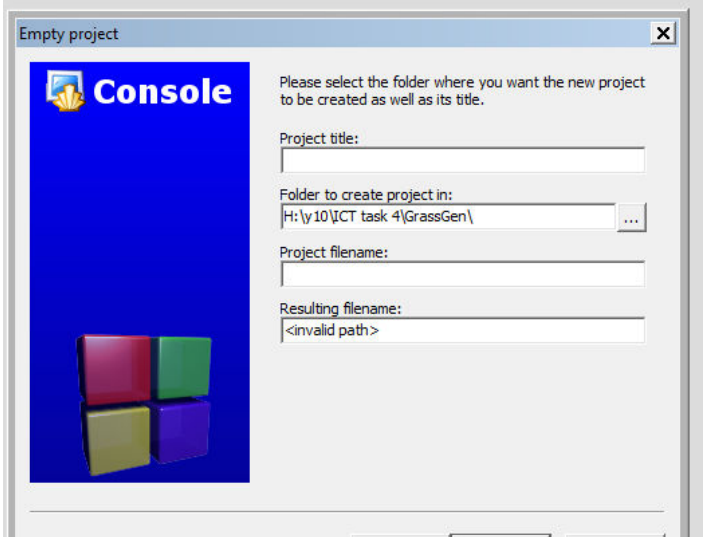
```
//! Define _IRR_COMPILE_WITH_JOYSTICK_SUPPORT_ if you want joystick events.
//#define _IRR_COMPILE_WITH_JOYSTICK_EVENTS_
```

I disabled software renderers/rasterizers/video drivers and the hardware DirectX, because I would have to download a 400mb SDK also it is windows only. The generator will not have a GUI or draw any triangles on the screen, so no video drivers are needed. However for testing purposes I will draw debug data with OpenGL. In the final version the OpenGL driver will be disabled and nothing will be drawn on screen. I also disabled the built-in irrlicht GUI for buttons, sliders etc. as they will not be needed.



Lastly I had to choose a "build target", I chose debug. It will be slower than the Release "fast-math" but it will print out a lot of debug data, warnings and errors helping me in my implementation. The final program will use a Release "fast -math" irrlicht with no OpenGL. I also chose to build a static library instead of a DLL, because I don't want the program to be in 2 pieces.

Setting up the Code::Blocks Project



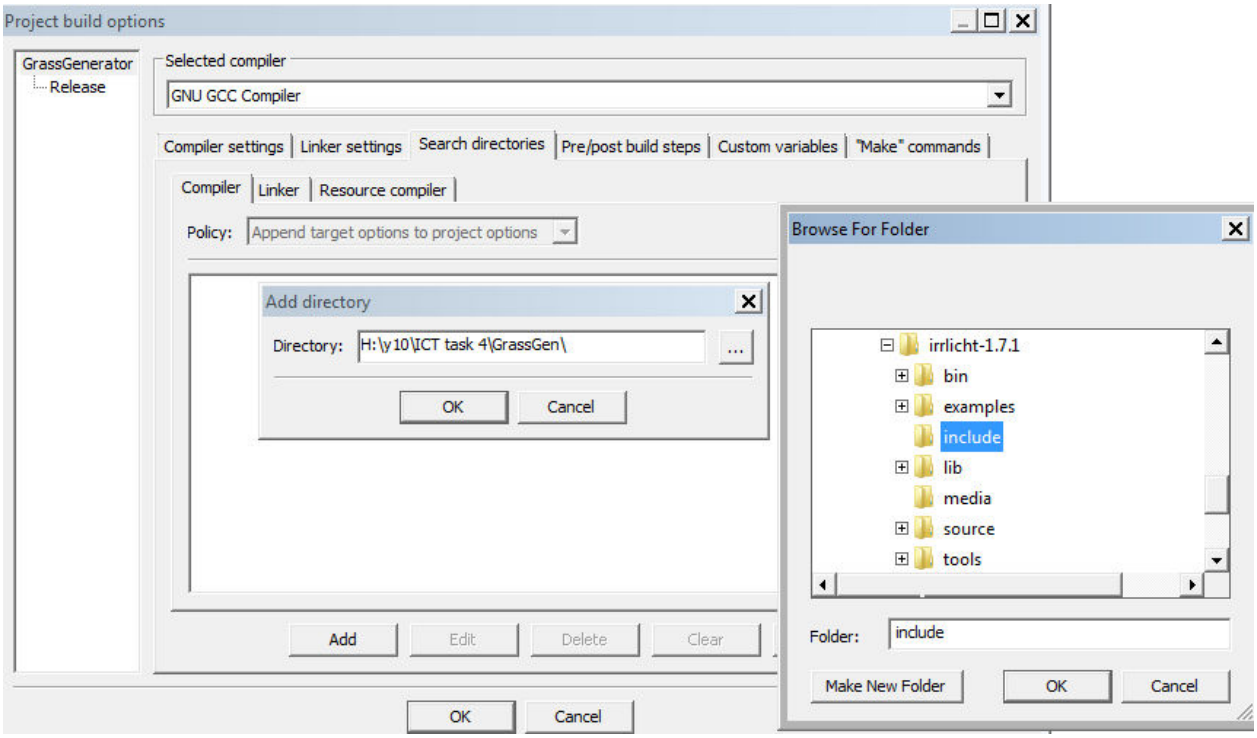
I use the project creation wizard to create an empty project in the right folder.

Then I had to create main.cpp make it create an IrrlichtDevice instance, sleep, close it and print a message to the command line. The engine should set up a window to draw in. I have typed up this C++ file.

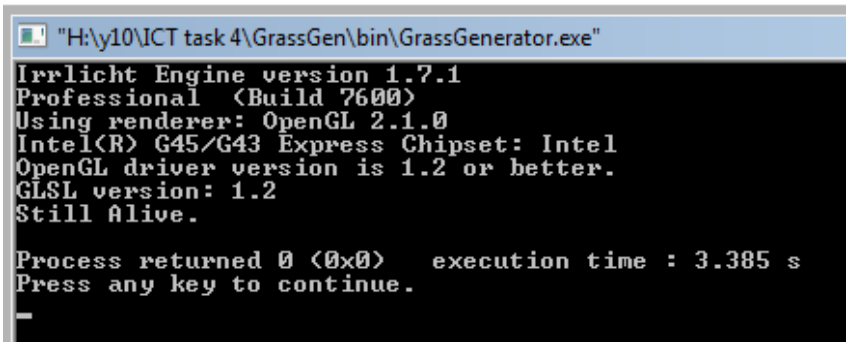
Note: I will explain the code in comments

```
// We need to include the file which defines all irrlicht's functions
// in order to be able to use them
#include <irrlicht.h>

/* namespaces are used by libraries to avoid naming clashes
for example one library may define a class called "GUIButton"
and if we use yet another one which defines a completely
different class with the same name, we will end up with errors.
therefore libraries define prefixes like "irr::" for irrlicht
to avoid writing out the prefix each time we state "using namespace irr;"*/
using namespace irr;

int main() // where the program starts
{
   /*Here I create a device and receive a pointer to it,
   the driver is OpenGL, and a window will be created in
   which all the graphics are drawn. The window size is
   800x600 pixels, with 24 bits of color depth*/
   IrrlichtDevice* device = createDevice(video::EDT_OPENGL,core::dimension2du(800,600));
   device->sleep(3000); // the program will pause/sleep for 3000 miliseconds = 3 seconds

   // here we close and delete the device
   device->closeDevice(); // we tell the device to start shutting down
   /*although device is a dynamically allocated object via "new"
   if implements a reference counting system, where references are
   incremented and decremented using grab() and drop(). The reference
   equals 1 on creation, if reference drops to 0, the object deletes
   itself. Therefore we cannot call "delete device;"*/
   device->drop();

   //We print a message to make sure the engine didnt crash
   printf("Still Alive.\n"); // The \n means enter/new line

   return 0; // the function needs to return a value to the OS
   //0 means successful termination
}
```

Of course I had to set up the options in the project in order to be able to find the irrlicht include files and link the irrlicht library.



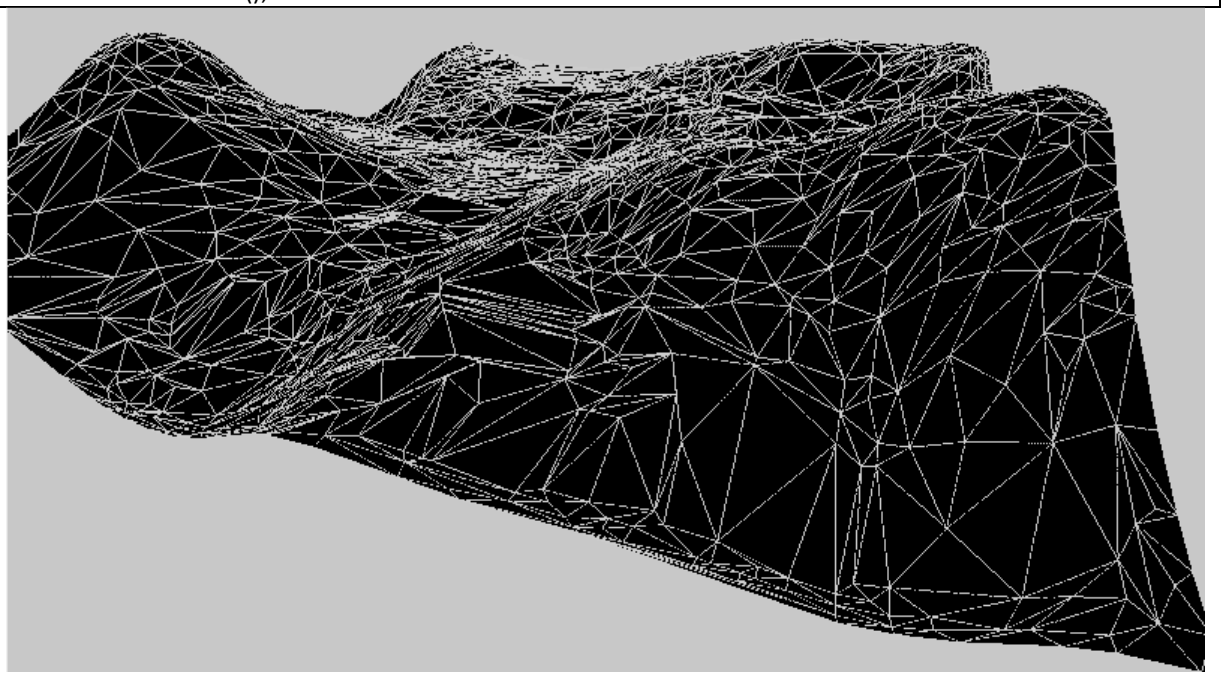
I also enabled optimization flags, which do not strip debugging symbols. The project built but it complained about not finding the irrlicht DLL, despite having compiled the lib as a static one. I just copied the irrlicht.dll from the examples folder from irrlicht and the program did what I expected.



Next I attempted to load the terrain model and display it

```
IrrlichtDevice* device = createDevice(video::EDT_OPENGL,core::dimension2du(800,600));
// Here I retrieve the handle to the VideoDriver interface which draws everything
video::IVideoDriver* driver = device->getVideoDriver();
// The scene manager loads models and draws the scene correctly (does frustum culling etc.)
scene::ISceneManager* smgr = device->getSceneManager();

// We load a mesh from a file
scene::IMesh* mesh = smgr->getMesh("../vegData/terrain.ms3d");
// The file may have not been loaded and the pointer can be null, so we need to check
if (mesh)
```

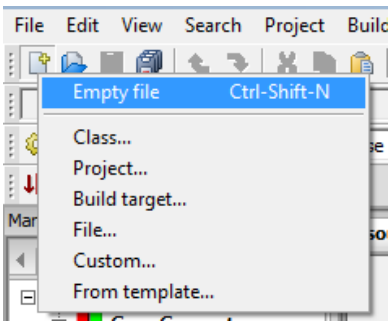
```
{
    scene::IMeshSceneNode* node = smgr->addMeshSceneNode(mesh);
    //We dont want just a black silhouette so we show a wireframe overlay to visualise
    //the individual triangles
    node->setDebugDataVisible(scene::EDS_MESH_WIRE_OVERLAY);
}

//We need a camera to be able to look around the model with the arrow keys
smgr->addCameraSceneNodeFPS(0,32.f,0.25);

while (device->run())
{
    //This flushes the whole window with a fully opaque greyish color RGB = (200,200,200)
    //It also enables us to draw
    driver->beginScene(true,true,video::SColor(255,200,200,200));
    smgr->drawAll();
    driver->endScene(); // Presents the buffer in the window - on Screen
}

device->closeDevice();
```



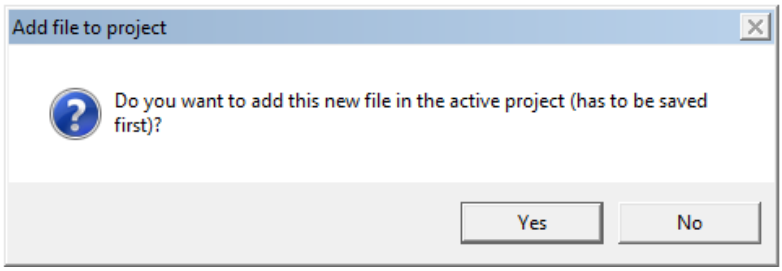
```
"H:\y10\ICT task 4\GrassGen\bin\GrassGenerator.exe"
Irrlicht Engine version 1.7.1
Professional  (Build 7600)
Using renderer: OpenGL 2.1.0
Intel(R) G45/G43 Express Chipset: Intel
OpenGL driver version is 1.2 or better.
GLSL version: 1.2
Loaded texture: H:/y10/ICT task 4/GrassGen/vegData/grassmap.png
Loaded mesh: ../vegData/terrain.ms3d
Resizing window (800 600)
Still Alive.

Process returned 0 (0x0)   execution time : 60.872 s
Press any key to continue.
```

Coding the GrassGenerator class

Here I create the interface which will provide us with the functionality of the program.



I added a C++ header file to the project in which I shall declare&&implement the GrassGenerator class.



I like to use the C* prefix notation for class headers (.h) and sources (.cpp), in this case GrassGenerator.h becomes CGrassGenerator (I use S for structs, E for enumerations). Also, I ran into a problem, in my design I intended on calling the class GrassGenerator and the namespace GrassGenerator. The namespace and the class name cannot be the same; therefore the class is now called CGrassGenerator.

CGrassGenerator.h

```
//guard block, to prevent the file being included twice
//if the definition is undefined the file has not been
//included. Then the definition is defined along with
//the class. Double definitions of classes and functions
//cause errors
#ifndef _C_GRASS_GENERATOR_H_
#define _C_GRASS_GENERATOR_H_

//despite having included it in main.cpp the
//CGrassGenerator.cpp won't see it so we
//include again
#include "irrlicht.h"

//enclose the class in a namespace, things inside the
//namespace become accessible only with GrassGenerator::
//in front or if "using namespace GrassGenerator;" is
//declared. A bit like irr::io::path
namespace GrassGenerator
{

//The class definition, the actual function code is
//contained in the implementation file (source, .cpp)
class CGrassGenerator
{
   //The following are private, therefore unique to each
   //instance of CGrassGenerator.
   private:
      float density;
      irr::c8 up_axis;
      irr::scene::ITriangleSelector* terrainModel;
```

```
    irr::video::IImage* vegetationMap;
    //we need to store some irrlicht subsystem's references
    irr::io::IFileSystem* fileSystem;
    irr::video::IVideoDriver* video;
    irr::scene::ISceneManager* smgr;

  //These functions can be accessed from the outside
  //i.e. the main() function. The load() and writeOut()
  //functions take const parameters because they are
  //references to avoid copying memory around and are
  //not allowed to change the original value.
  public:
    //The constructor sets up the access to irrlicht's
    //functions
    CGrassGenerator(irr::IrrlichtDevice* irrlicht)
    {
      //check for invalid irrlicht
      if (irrlicht)
      {
        fileSystem = irrlicht->getFileSystem();
        video = irrlicht->getVideoDriver();
        smgr = irrlicht->getSceneManager();
      }
    }
    bool load(const irr::io::path& xmlFilename);
    void process();
    bool writeOut(const irr::io::path& Filename);
};

}

#endif // _C_GRASS_GENERATOR_H_
```

main.cpp

```
#include <irrlicht.h>
//need to include the definition of the new class in the program
#include "CGrassGenerator.h"

using namespace irr;
```

CGrassGenerator.cpp

```
//This is the implementation file of the CGrassGenerator class.
//The compiler creates a .o file for each implementation file and
//links the into a program or library

//We need to include the declarations
#include "CGrassGenerator.h"

//we can “use” a namespace now because it will not carry on
//into more source files (unlike from headers)
using namespace irr;

//Again we need to enclose everything in a namespace
namespace GrassGenerator
{
```

```
//This functions loads the input parameters from an XML file
bool CGrassGenerator::load(const irr::io::path& xmlFilename)
{
   //The function doesn't do anything yet.
   density=0;
   up_axis=0;
   terrainModel=0;
   vegetationMap=0;
   return true;
}

//The two functions are empty for now, because they dont do anything
void CGrassGenerator::process()
{
   return;
}
bool CGrassGenerator::writeOut(const irr::io::path& Filename)
{
   return false;
}

} // namespace GrassGenerator
```

After these additions the program compiled successfully and has not increased in size (still 21kb).

Loading parameters from XML

CGrassGenerator.cpp

```
bool CGrassGenerator::load(const irr::io::path& xmlFilename)
{
   //If we cannot use the fileSystem pointer then the function
   //cannot continue, therefore it fails (success=false) and
   //quits
   if (!fileSystem)
   {
      printf("GRASS GENERATOR ERROR: The IFileSystem pointer is NULL.\n");
      return false;
   }

   io::IXMLReader* xmlReader = fileSystem->createXMLReader(xmlFilename);
   //if the file doesnt exist or the irrlicht function fails, it
   //returns a NULL pointer which is unsafe to use and the loading
   //cannot continue
   if (!xmlReader)
   {
      printf("GRASS GENERATOR ERROR: The IXMLReader pointer is NULL, file most probably doesn't exist.\n");
      return false;
   }
   //A class which simplifies reading parameters from an XML file
   io::IAttributes* attrs = fileSystem->createEmptyAttributes();
   //If the fileSystem function fails it returns a NULL pointer
   if (!attrs)
   {
      printf("GRASS GENERATOR ERROR: The IAttributes pointer is NULL.\n");
      //The instance of IXMLReader is valid and will cause a memory
      //leak if not disposed of
      xmlReader->drop();
      return false;
   }
   //Loads just the global parameters
   attrs->read(xmlReader,false,L"grassGeneratorParameters");
   //Delete/dispose of the instances of xmlReader and attrs
   attrs->drop();
   xmlReader->drop();

   density=0;
   up_axis=0;
   terrainModel=0;
   vegetationMap=0;
   return true;
}
```

CGrassGenerator.h

```
//guard block, to prevent the file being included twice
//if the definition is undefined the file has not been
//included. Then the definition is defined along with
//the class. Double definitions of classes and functions
//cause errors
#ifndef _C_GRASS_GENERATOR_H_
#define _C_GRASS_GENERATOR_H_

#include "irrlicht.h"

//enclose the class in a namespace, things inside the
//namespace become accessible only with GrassGenerator::
//in front or if "using namespace GrassGenerator;" is
//declared. A bit like irr::io::path
namespace GrassGenerator
{

//The class definition, the actual function code is
//contained in the implementation file (source, .cpp)
class CGrassGenerator
{
   //The following are private, therefore unique to each
   //instance of CGrassGenerator. They are also declared
   //constant so the data cannot be modified once set.
   private:
      float density;
      irr::c8 up_axis;
      irr::scene::ITriangleSelector* terrainModel;
      irr::video::IImage* vegetationMap;
      //we need to store some irrlicht subsystem's references
      irr::io::IFileSystem* fileSystem;
      irr::video::IVideoDriver* video;
      irr::scene::ISceneManager* smgr;

   //These functions can be accessed from the outside
   //i.e. the main() function. The load() and writeOut()
   //functions take const parameters because they are
   //references to avoid copying memory around and are
   //not allowed to change the original value.
   public:
      //The constructor sets up the access to irrlicht's
      //functions
      CGrassGenerator(irr::IrrlichtDevice* irrlicht)
      {
         //check for invalid irrlicht
         if (irrlicht)
         {
            fileSystem = irrlicht->getFileSystem();
            video = irrlicht->getVideoDriver();
            smgr = irrlicht->getSceneManager();
         }
      }
      bool load(const irr::io::path& xmlFilename);
      void process();
      bool writeOut(const irr::io::path& Filename);
```

```
};

}

#endif // _C_GRASS_GENERATOR_H_
```

Main.cpp

```
#include "CGrassGenerator.h"

using namespace irr;
using namespace GrassGenerator;

//the modified main() function which allows to input parameters like the path to the input XML
int main(int argc,char** argv)
{
   IrrlichtDevice* device = createDevice(video::EDT_OPENGL,core::dimension2du(800,600));
   // Here I retrieve the handle to the VideoDriver interface which draws everything
   video::IVideoDriver* driver = device->getVideoDriver();
   // The scene manager loads models and draws the scene correctly (does frustum culling etc.)
   scene::ISceneManager* smgr = device->getSceneManager();

   //Here we create a DYNAMICALLY allocated instance of the grass generator
   CGrassGenerator* grassGenInstance = new CGrassGenerator(device);

   //Here the program checks the input arguments from the command line, if they are not valid
   //it will use the default path to input file "./input.xml"
   //The validation rules are: presence check, length check, file existence check
   core::stringc inputXMLPath = "./input.xml";
   if (argc>0) //If there are any arguments
   {
      core::stringc temp = core::stringc(argv[1]);
      if (temp.size()>4) //has a file extension
      {
         if (device->getFileSystem()->existFile(temp))
            inputXMLPath=temp;
      }
   }

   //Here we load the parameters
   grassGenInstance->load(inputXMLPath);

...

   //must dispose of the dynamic instance or memory leak
   delete grassGenInstance;
```

Included a bad program argument, to file which doesn't exist.

And then amended it to "../vegData/input.xml", and then with no argument at all (output same as in the first case).

```
Irrlicht Engine version 1.7.1
Professional  (Build 7600)
Using renderer: OpenGL 2.1.0
Intel(R) G45/G43 Express Chipset: Intel
OpenGL driver version is 1.2 or better.
GLSL version: 1.2
GRASS GENERATOR ERROR: The IXMLReader pointer is NULL, file most probably doesn'
t exist.
Loaded texture: H:/y10/ICT task 4 prog/GrassGen/vegData/grassmap.png
Loaded mesh: ../vegData/terrain.ms3d
Resizing window (800 600)
Still Alive.
```

```
Irrlicht Engine version 1.7.1
Professional  (Build 7600)
Using renderer: OpenGL 2.1.0
Intel(R) G45/G43 Express Chipset: Intel
OpenGL driver version is 1.2 or better.
GLSL version: 1.2
Loaded texture: H:/y10/ICT task 4/GrassGen/vegData/grassmap.png
Loaded mesh: ../vegData/terrain.ms3d
Resizing window (800 600)
Still Alive.
```

Unplanned Changes

I decided to abandon IAttributes to read data from the xml because the xml would have to be laid out differently, so instead I will program my own routine to load the parameters. Firstly I had to amend the input.xml file because the paths to my test data were incorrect. Also I made one change to the layout. I decided not to enclose the Vegetation Type declarations in <vegetationTypes> and </vegetationTypes> because the xml reader had some problems with recognising the nodes enclosed. Instead I resorted to a simple comment <!—vegetationTypes -->

```
<?xml version="1.0"?>
<grassGeneratorParameters>
  <!--This is a parameters file for the Grass Generator. Text enclosed in !—and – is a comment -->
  <terrainModel file="./terrain.ms3d" />
  <vegetationMap  file="./grassmap.png" />
  <globals density=”0.1” up_axis=”Y” />
  <!--vegetationTypes -->
      <vegetationType colorMask=”g” ratio=”36” scale=”2.45” position_var=”0.5” scale_var=”0.068”
model=”./vegData/grass1.ms3d” texture=”./vegData/grass.png”  />
      <vegetationType colorMask=”rg” ratio=”24” scale=”2.45” position_var=”0.5” scale_var=”0.068”
model=”./vegData/grass1.ms3d” texture=”./vegData/grass_2.png”  />

.. and so on
```

Next I have amended the CGrassGenerator::load function to load the inputs. The code handles cases when the input file is erroneous. This is how I got to know that the paths are not relative to the input file.

CGrassGenerator.cpp

```
//This functions loads the input parameters from an XML file
bool CGrassGenerator::load(const irr::io::path& xmlFilename)
{
   density = 1.f;
```

```
up_axis = 1;
//If we cannot use the fileSystem pointer then the function
//cannot continue, therefore it fails (success=false) and
//quits
if (!fileSystem)
{
   printf("GRASS GENERATOR ERROR: The IFileSystem pointer is NULL.\n");
   return false;
}

io::IXMLReader* xmlReader = fileSystem->createXMLReader(xmlFilename);
//if the file doesnt exist or the irrlicht function fails, it
//returns a NULL pointer which is unsafe to use and the loading
//cannot continue
if (!xmlReader)
{
   printf("GRASS GENERATOR ERROR: The IXMLReader pointer is NULL, file most probably doesn't exist.\n");
   return false;
}

core::stringc tmp = xmlFilename;
u32 length = xmlFilename.size()-core::deletePathFromFilename(tmp).size();
fileSystem->addFolderFileArchive( xmlFilename.subString(0,length).c_str() );

core::stringc terrainModelFilename;
core::stringc vegetationMapFilename;
while (xmlReader->read()) // returns false if end of file is reached
{
   if (xmlReader->getNodeType()!=io::EXN_COMMENT)
   {
      core::stringw nodeName = xmlReader->getNodeName();
      if (nodeName==L"terrainModel")
      {
         terrainModelFilename = xmlReader->getAttributeValueSafe(L"file");
      }
      else if (nodeName==L"vegetationMap")
         vegetationMapFilename = xmlReader->getAttributeValueSafe(L"file");
      else if (nodeName==L"globals")
      {
         density = xmlReader->getAttributeValueAsFloat(L"density");
         if (density<=0.f) // cannot have density 0.f or negative
         {
            printf("GRASS GENERATOR ERROR: The density parameter is invalid.\n");
            xmlReader->drop();
            return false;
         }
         core::stringc upx = xmlReader->getAttributeValueSafe(L"up_axis");
         if (upx.size()!=1) // only one character in length
         {
            printf("GRASS GENERATOR ERROR: The up axis is not declared.\n");
            xmlReader->drop();
            return false;
         }
         up_axis = 127; //wont get changed if up_axis doesnt equal x,X,y,Y,z or Z
         if (upx[0]=='X'||upx[0]=='x')
            up_axis = 0;
         else if (upx[0]=='Y'||upx[0]=='y')
```

```
            up_axis = 1;
         else if (upx[0]=='Z'||upx[0]=='z')
            up_axis = 2;
         if (up_axis==127) // not changed
         {
            printf("GRASS GENERATOR ERROR: The up axis is not declared.\n");
            xmlReader->drop();
            return false;
         }
      }
   }
   //Dont need the xmlReader anymore
   xmlReader->drop();

   //Load the terrain model
   scene::IMesh* mesh = smgr->getMesh(terrainModelFilename);
   if (!mesh)
   {
      printf("GRASS GENERATOR ERROR: The terrain IMesh pointer is NULL, file most probably doesn't
exist.\n");
      return false;
   }
   terrainNode = smgr->addMeshSceneNode(mesh);
   terrainModel = smgr->createOctreeTriangleSelector(mesh,0,32);

   //Load the Vegetation Map
   vegetationMap = video->createImageFromFile(vegetationMapFilename);
   if (!vegetationMap)
   {
      printf("GRASS GENERATOR ERROR: The vegetationMap IImage pointer is NULL, file most probably doesn't
exist.\n");
      return false;
   }

   return true;
}
```

I turned out that I needed an ISceneNode in order to create an instance of OctreeTriangleSelector. I had to add a few helper functions to be able to retrieve the data and visualise it.

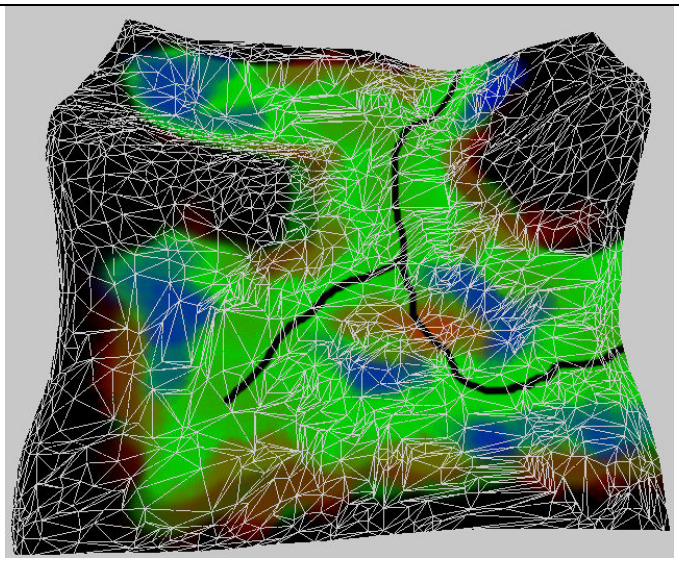
CGrassGenerator.h

```
class CGrassGenerator
{
   //The following are private, therefore unique to each
   //instance of CGrassGenerator. They are also declared
   //constant so the data cannot be modified once set.
   private:
      float density;
      irr::c8 up_axis;
      irr::scene::ITriangleSelector* terrainModel;
      irr::scene::ISceneNode* terrainNode;
      irr::video::IImage* vegetationMap;
      //we need to store some irrlicht subsystem's references
      irr::io::IFileSystem* fileSystem;
      irr::video::IVideoDriver* video;
```

```
    irr::scene::ISceneManager* smgr;

  //These functions can be accessed from the outside
  //i.e. the main() function. The load() and writeOut()
  //functions take const parameters because they are
  //references to avoid copying memory around and are
  //not allowed to change the original value.
  public:
    //The constructor sets up the access to irrlicht's
    //functions
    CGrassGenerator(irr::IrrlichtDevice* irrlicht)
    {
      //check for invalid irrlicht
      if (irrlicht)
      {
        fileSystem = irrlicht->getFileSystem();
        video = irrlicht->getVideoDriver();
        smgr = irrlicht->getSceneManager();
      }
    }
    bool load(const irr::io::path& xmlFilename);
    void process();
    bool writeOut(const irr::io::path& Filename);
    irr::scene::ISceneNode* getTerrainNode() {return terrainNode;}
    irr::video::IImage* getVegetationMap() {return vegetationMap;}
};
```

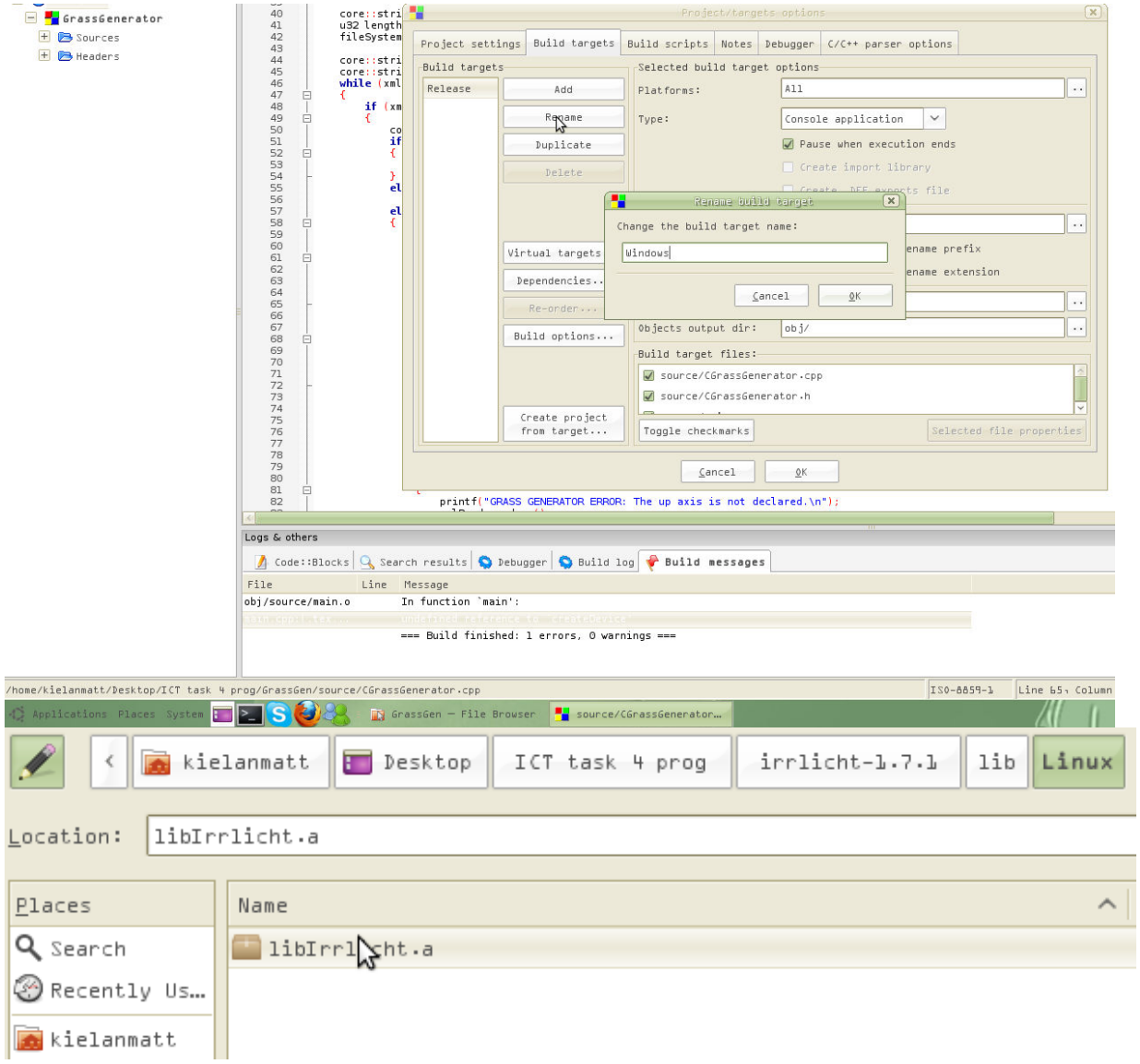
main.cpp

```
The previous part of the file has not been modified

  //Here we load the parameters
  grassGenInstance->load(inputXMLPath);

  if (grassGenInstance->getTerrainNode())
  {
    grassGenInstance->getTerrainNode()->setDebugDataVisible(scene::EDS_MESH_WIRE_OVERLAY);
    grassGenInstance->getTerrainNode()->setMaterialFlag(video::EMF_LIGHTING,false);
    grassGenInstance->getTerrainNode()->setMaterialTexture(0,driver-
>addTexture("VMap",grassGenInstance->getVegetationMap()));
  }
```

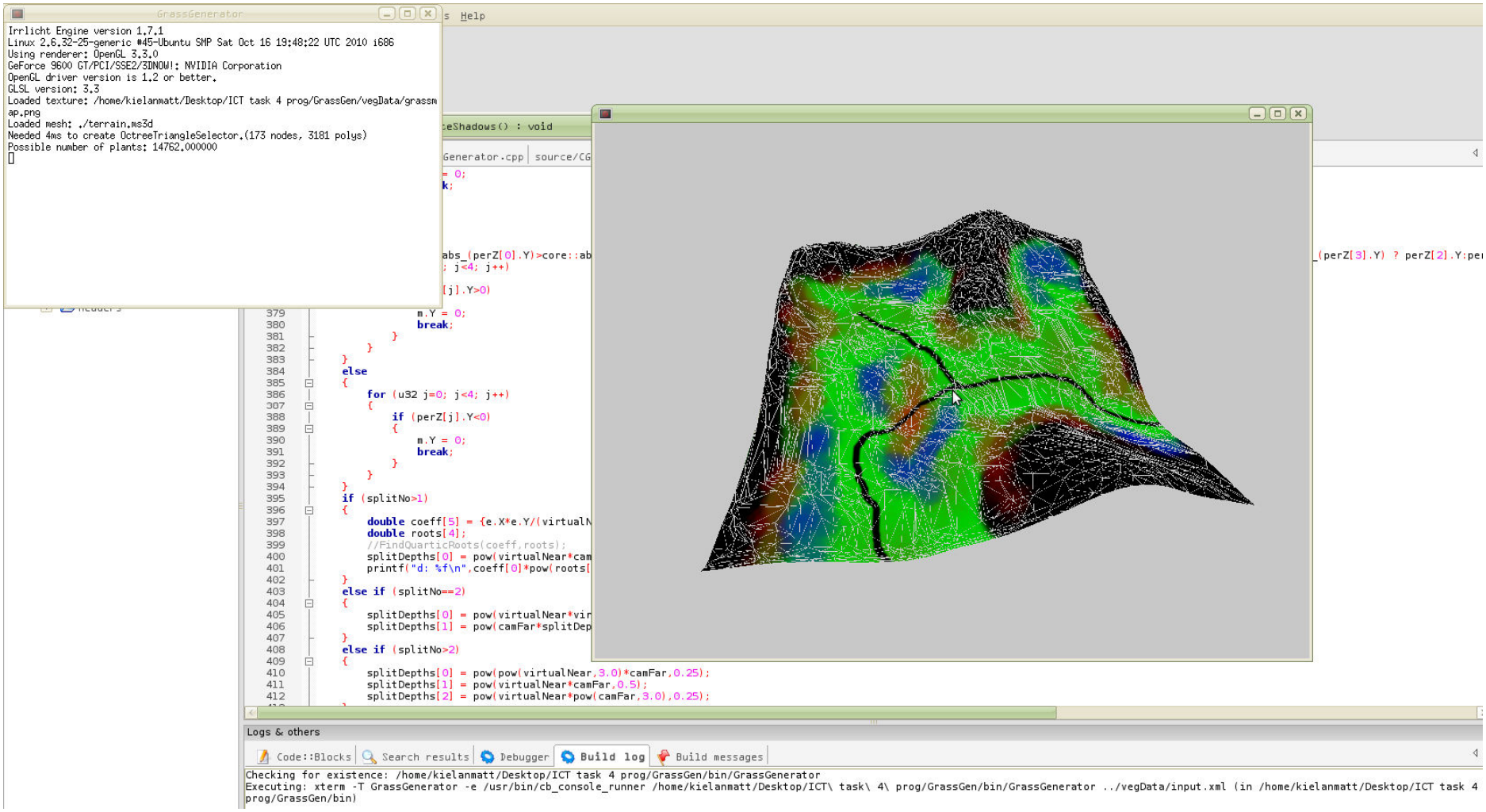


Linux Version

One of my client requirements was to ensure the system runs on linux, I took the source home on my pendrive and attempted to compile it on my home computer. I had to build the irrlicht engine and add another build target entitled “Linux” to my application build target.



Most of the settings stayed the same but I had to link to a different build of irrlicht and a few additional libraries. The whole process was relatively painless and ended with success as the next screenshot shows.



NOTE: The sourcecode in the background is from another project.

Beginning Processing

I have not yet troubled myself with loading the Vegetation Type declarations, right now I am moving on onto processing and I will tackle the problem of generating initial plant positions. The first task is to estimate the total number of plants that will be generated and the total number of vertices (which will require for me to go back to the loading function and load the Vegetation Types).

```
GLSL version: 1.2
Loaded texture: h:/y10/ict task 4 prog/grassgen/vegdata/grassmap.png
Loaded mesh: ./terrain.ms3d
Needed 3ms to create OctreeTriangleSelector.(177 nodes, 3181 polys)
Possible number of plants: 14762.000000
.Resizing window (800 600)
Still Alive.

Process returned 0 (0x0)   execution time : 7.363 s
Press any key to continue.
_
```

The next task is to go through the vegetation map and prune away the plant positions depending on alpha channel value.

CGrassGenerator.cpp

```
//The processing function
void CGrassGenerator::process()
{
   core::aabbox3df bbox = terrainNode->getBoundingBox();
   core::vector2df scale,translation;
   // here we handle which axis is “up” X,Y, or Z
   if (up_axis==0)
   {
      scale.Y = 1.f/(bbox.MaxEdge.Y-bbox.MinEdge.Y);
      scale.X = 1.f/(bbox.MaxEdge.Z-bbox.MinEdge.Z);
      translation.Y = (bbox.MinEdge.Y)*scale.Y;
      translation.X = (bbox.MinEdge.Z)*scale.X;
   }
   else if (up_axis==1)
   {
      scale.Y = 1.f/(bbox.MaxEdge.Z-bbox.MinEdge.Z);
      scale.X = 1.f/(bbox.MaxEdge.X-bbox.MinEdge.X);
      translation.Y = (bbox.MinEdge.Z)*scale.Y;
      translation.X = (bbox.MinEdge.X)*scale.X;
   }
   else
   {
      scale.Y = 1.f/(bbox.MaxEdge.Y-bbox.MinEdge.Y);
      scale.X = 1.f/(bbox.MaxEdge.X-bbox.MinEdge.X);
      translation.Y = (bbox.MinEdge.Y)*scale.Y;
```

```
        translation.X = (bbox.MinEdge.X)*scale.X;
    }
    core::vector2df numberOfGrids(core::vector2df(1.f)/(scale*density));
    numberOfGrids.X = floor(numberOfGrids.X);
    numberOfGrids.Y = floor(numberOfGrids.Y);
    //we don't want a row of plants at x,0 or 0,y. Basically skip the first row
    numberOfGrids -= 1.f;

    //No plants in the middle (too wide spacing)
    if (numberOfGrids.X<1||numberOfGrids.Y<1)
    {
        printf("GRASS GENERATOR ERROR: The density parameter is too large.\n");
        return;
    }

    //8bit data for each plant position (1 for red VT,2 for grass and 3 for blue, 0 for no plant)
    //Dynamically allocated array in order to control memory use
    c8* positionColours = new c8[u32(numberOfGrids.X*numberOfGrids.Y)];

    //so we modify the right place in the above array
    u32 arrayOffset = numberOfGrids.X+1;
    core::vector2df sizeConstant(vegetationMap->getDimension().Width,vegetationMap-
>getDimension().Height);
    sizeConstant /= (numberOfGrids+1.f);
    u32 no_of_plants = 0;

    for (f32 y=1.f; y<=numberOfGrids.Y; y++)
    {
        //y pixel position only changes when the row changes
        float ypixel = floorf(y*sizeConstant.Y);
        // we compute half of the array index here so we don't do X*Y number of operations but Y
        u32 arrayIndex = u32(y*numberOfGrids.X-arrayOffset);
        for (f32 x=1.f; x<=numberOfGrids.X; x++)
        {
            //exact pixel position
            float m = x*sizeConstant.X;
            //round down to get the physical pixel
            float xpixel = floorf(m);
            //get the decimal part of the exact pixel position (percentage of the way between pixels)
            float mu = core::fract(m);
            video::SColor temp = vegetationMap->getPixel(xpixel,ypixel);
            // simple discard threshold, will replace with a stratified random function
            if (temp.getAlpha()>127)
            {
                //no plant type determination yet
                positionColours[arrayIndex+x] = 1;
                no_of_plants++;
            }
        }
    }
```

```
   printf("Possible number of plants: %u \n.",no_of_plants);
   //Delete the dynamically allocated array
   delete [] positionColours;

   return;
}
```

With the threshold of 127 (50%) of opacity there are only 8712 plant positions as opposed to 14762. I carried out a series of micro-tests, here is a table:

Value	10%	20%	30%	40%	50%	60%	70%	80%	90%
Result	11322	10643	10081	9406	8712	7738	6527	5084	3478

I have then modified the function to randomly generate the threshold and the amount of "would be generated" plants oscillated around 7500 to 8100 (like a 50% threshold). This proves that the sampling method is more or less statistically correct.

```
for (f32 y=1.f; y<=numberOfGrids.Y; y++)
   {
      float ypixel = floorf(y*sizeConstant.Y);
      u32 arrayIndex = u32(y*numberOfGrids.X-arrayOffset);
      srand(y*time(NULL));
      for (f32 x=1.f; x<=numberOfGrids.X; x++)
      {
         float m = x*sizeConstant.X;
         float xpixel = floorf(m);
         float mu = core::fract(m);
         video::SColor temp = vegetationMap->getPixel(xpixel,ypixel);
         if (temp.getAlpha()>=(rand()%256))
         {
            srand(x*time(NULL));
            positionColours[arrayIndex+u32(x)] = 1;
            no_of_plants++;
         }
      }
   }
```

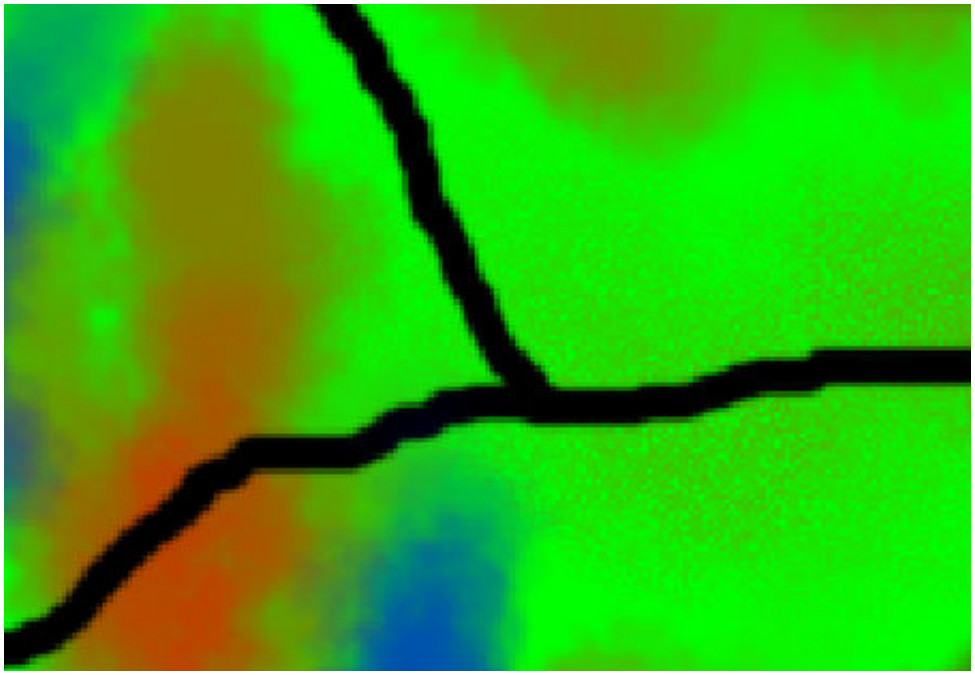
As I have mentioned in my design I am going to use multithreading to speed up the grass generation by using all the CPU cores available (or hyperthreading), dual core computers have two cores and some have hyperthreading (4 hardware cores) while the system may even be used on quad cores. Therefore I have assumed that the best number of threads to use is 4, including the main program thread. This works out nicely as I can split the above function over 4 threads and split the vegetation map into 4 smaller squares. I am not very proficient with win32 API threading so I am going to use pthreads for windows instead (pthreads are a posix/unix standard) which is basically a library with almost the same functionality (function names etc.) which does its work in win32 API threads (kind of translator). This will save me the effort to code 2 different ways of threading for each of my target OSes.

```
//a thread function
#define NUM_THREADS     5
void *PrintHello(void *threadid)
{
   long tid;
   tid = *((long*)threadid);
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

...

I stuck the following onto the processing function

printf("Possible number of plants: %u \n.",no_of_plants);

    delete [] positionColours;

    pthread_t threads[NUM_THREADS];
    int rc;
    long t[NUM_THREADS];
    for(long i=0; i<NUM_THREADS; i++){
       t[i] = i;
       printf("In main: creating thread %ld\n", i);
       rc = pthread_create(&threads[i], NULL, PrintHello, (void *)(t+i));
       if (rc){
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          return;
       }
    }
    for(long i=0; i<NUM_THREADS; i++)
       (void) pthread_join(threads[i],NULL);

    return;
}
```

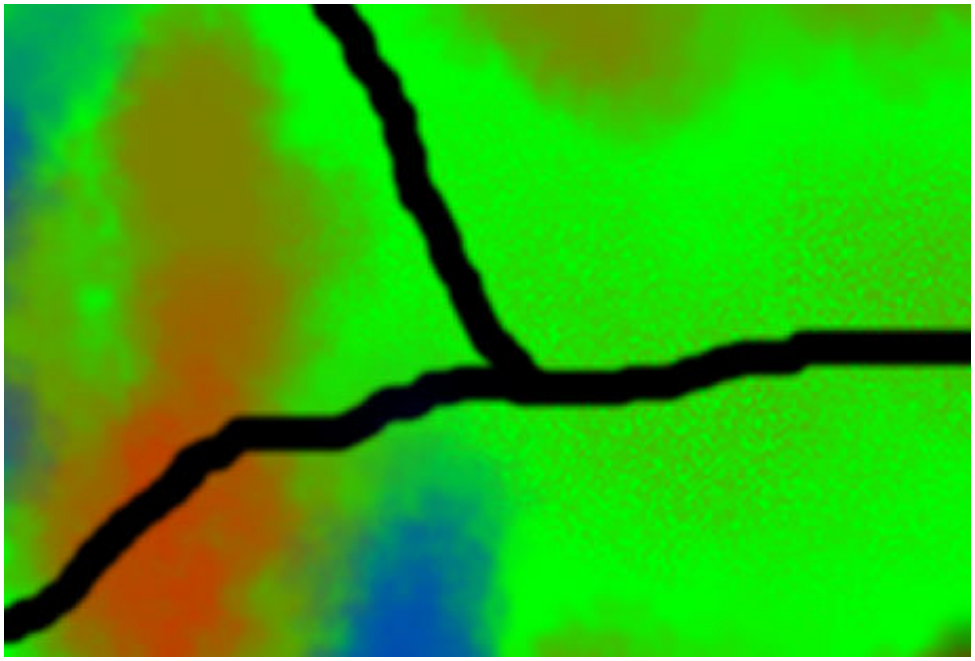
```
Professional  (Build 7600)
Using renderer: OpenGL 2.1.0
Intel(R) G45/G43 Express Chipset: Intel
OpenGL driver version is 1.2 or better.
GLSL version: 1.2
Loaded texture: h:/y10/ict task 4 prog/grassgen/vegdata/grassmap.png
Loaded mesh: ./terrain.ms3d
Needed 2ms to create OctreeTriangleSelector.(177 nodes, 3181 polys)
Possible number of plants: 7616
.In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
In main: creating thread 3
Hello World! It's me, thread #1!
In main: creating thread 4
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
Hello World! It's me, thread #3!
Resizing window (800 600)
Still Alive.

Process returned 0 (0x0)   execution time : 2.075 s
Press any key to continue.
```

For convenience I tackled bicubic interpolation and threading at once. The test to show the results took a bit of thought to come up with. First I need more plant positions than there are pixels in the vegetation map so the information is under sampled, then I save the colours sampled at the position of each plant into the shadow map. I decided to thread the function straight away, I created a buffer of pixels (one per plant position) and each thread only modified a quarter of the buffer. Splitting the image processing into 4 roughly equal tiles.

Here is no interpolation:



And bicubicly interpolated:



And here is the code:

CGrassGenerator.h

```
//invisible functions and data types (just in this .cpp file)

//Different instance of the struct for each thread
struct imageMarchThreadInput
{
   //Amount of plants in X and Y
   core::vector2df noOfGrids;
   video::IImage* vegMap;
   //Colour data buffer for each plant
   //Will shrink just to the Vegetation Type ID
   video::SColorf* posColours;
   u8* posColoursw; //temporary data for the bicubic interpolation test
   //A bounding rectangle (area of the buffer/test image for the thread to process)
   core::rectf threadoffset;
};

//thread function
void *InterpolatePositions(void *iMTI)
{
   //safety check
   if (!iMTI)
      pthread_exit(NULL);

   //typecasting the input
   imageMarchThreadInput td;
   td = *((imageMarchThreadInput*)iMTI);
   core::vector2df numberOfGrids = td.noOfGrids;
   video::IImage* vegetationMap = td.vegMap;
   video::SColorf* positionColours = td.posColours;
   u8* positionColoursw = td.posColoursw;

   //grid position and array position don't line up exactly, hence the offset
   u32 arrayOffset = numberOfGrids.X+1;
   //Here I calculate the size offset, any grid position multiplied by it will give a
   //corresponding texture coordinate on the vegetation map in (0-image_size) range
   core::vector2df sizeConstant(vegetationMap->getDimension().Width,vegetationMap-
>getDimension().Height);
   sizeConstant /= (numberOfGrids+1.f);

   for (f32 y=td.threadoffset.UpperLeftCorner.Y; y<=td.threadoffset.LowerRightCorner.Y; y++)
   {
      //image space y position
      float my = y*sizeConstant.Y;
      //image space y position rounded down to nearest pixel
      float ypixel = floorf(my);
      //offset in the test image data array
      u32 arrayIndex = u32(y*numberOfGrids.X-arrayOffset);
       for (f32 x=td.threadoffset.UpperLeftCorner.X; x<=td.threadoffset.LowerRightCorner.X; x++)
```

```
    {
        //image space x position
        float m = x*sizeConstant.X;
        //the x position rounded to nearest pixel
        u32 xpixel = floorf(m);
        //percentage of the way from the previous pixel to next as a decimal (fractal part)
        float mu = core::fract(m);
        //the same value squared
        float mu2 = mu*mu;
        //the same value cubed
        float mu3 = mu*mu2;
        //bicubically interpolated column of 4 pixels, 2 on each side of the desired position
        video::SColorf temp[4];
        for (s32 i=-1; i<3; i++)
        {
            //the interpolation algorithm, for each of the column pixels an entire row has to be
            //interpolated
            u32 ypixeli = ypixel+i;
            video::SColorf y1 = video::SColorf(vegetationMap->getPixel(xpixel,ypixeli));
            video::SColorf y0 = video::SColorf(vegetationMap->getPixel(xpixel-1,ypixeli));
            video::SColorf y2 = video::SColorf(vegetationMap->getPixel(xpixel+1,ypixeli));
            video::SColorf y3 = video::SColorf(vegetationMap->getPixel(xpixel+2,ypixeli));
            y3.r -= y2.r + y0.r - y1.r;
            y3.g -= y2.g + y0.g - y1.g;
            y3.b -= y2.b + y0.b - y1.b;
            y3.a -= y2.a + y0.a - y1.a;
            y2.r -= y0.r;
            y2.g -= y0.g;
            y2.b -= y0.b;
            y2.a -= y0.a;
            y0.r -= y1.r + y3.r;
            y0.g -= y1.g + y3.g;
            y0.b -= y1.b + y3.b;
            y0.a -= y1.a + y3.a;

temp[i+1].set(y3.a*mu3+y0.a*mu2+y2.a*mu+y1.a,y3.r*mu3+y0.r*mu2+y2.r*mu+y1.r,y3.g*mu3+y0.
g*mu2+y2.g*mu+y1.g,y3.b*mu3+y0.b*mu2+y2.b*mu+y1.b);
        }
        //percentage of the way in the y axis
        mu = core::fract(my);
        //same operations as with interpolating a row
        mu2 = mu*mu;
        mu3 = mu*mu2;
        temp[3].r -= temp[2].r + temp[0].r - temp[1].r;
        temp[3].g -= temp[2].g + temp[0].g - temp[1].g;
        temp[3].b -= temp[2].b + temp[0].b - temp[1].b;
        temp[3].a -= temp[2].a + temp[0].a - temp[1].a;
        temp[2].r -= temp[0].r;
        temp[2].g -= temp[0].g;
        temp[2].b -= temp[0].b;
        temp[2].a -= temp[0].a;
```

```
        temp[0].r -= temp[1].r + temp[3].r;
        temp[0].g -= temp[1].g + temp[3].g;
        temp[0].b -= temp[1].b + temp[3].b;
        temp[0].a -= temp[1].a + temp[3].a;

        //the exact floating point values for rgba at each plant position
positionColours[arrayIndex+u32(x)].set(temp[3].a*mu3+temp[0].a*mu2+temp[2].a*mu+temp[1].a,t
emp[3].r*mu3+temp[0].r*mu2+temp[2].r*mu+temp[1].r,temp[3].g*mu3+temp[0].g*mu2+temp[2].
g*mu+temp[1].g,temp[3].b*mu3+temp[0].b*mu2+temp[2].b*mu+temp[1].b);
        //because of the way integer numbers are stored and operated on, if the number is beyond
        //the type's range, the number value repeat. In this case I use an unsigned char, which can
        //represent numbers from 0 to 255, if a number is less than 0 then the computer starts
        //counting down from 255 again and if the number is greater than 255 the computer starts
        //counting up from 0 again. The bicubic interpolation tends to extrapolate sometimes, it took
        //me quite a while to figure out the cause of random white and black halos around the black
        //path in the vegetation map image
        positionColoursw[(arrayIndex+u32(x))*4] =
core::clamp(positionColours[arrayIndex+u32(x)].b*255,0.f,255.f);
        positionColoursw[(arrayIndex+u32(x))*4+1] =
core::clamp(positionColours[arrayIndex+u32(x)].g*255,0.f,255.f);
        positionColoursw[(arrayIndex+u32(x))*4+2] =
core::clamp(positionColours[arrayIndex+u32(x)].r*255,0.f,255.f);
        positionColoursw[(arrayIndex+u32(x))*4+3] =
core::clamp(positionColours[arrayIndex+u32(x)].a*255,0.f,255.f);
      }
   }

  pthread_exit(NULL);
}

//The processing function
void CGrassGenerator::process()
{
   core::aabbox3df bbox = terrainNode->getBoundingBox();
   core::vector2df scale,translation;
   if (up_axis==0)
   {
      scale.Y = 1.f/(bbox.MaxEdge.Y-bbox.MinEdge.Y);
      scale.X = 1.f/(bbox.MaxEdge.Z-bbox.MinEdge.Z);
      translation.Y = (bbox.MinEdge.Y)*scale.Y;
      translation.X = (bbox.MinEdge.Z)*scale.X;
   }
   else if (up_axis==1)
   {
      scale.Y = 1.f/(bbox.MaxEdge.Z-bbox.MinEdge.Z);
      scale.X = 1.f/(bbox.MaxEdge.X-bbox.MinEdge.X);
      translation.Y = (bbox.MinEdge.Z)*scale.Y;
      translation.X = (bbox.MinEdge.X)*scale.X;
   }
   else
   {
```

```
        scale.Y = 1.f/(bbox.MaxEdge.Y-bbox.MinEdge.Y);
        scale.X = 1.f/(bbox.MaxEdge.X-bbox.MinEdge.X);
        translation.Y = (bbox.MinEdge.Y)*scale.Y;
        translation.X = (bbox.MinEdge.X)*scale.X;
    }
    core::vector2df numberOfGrids(core::vector2df(1.f)/(scale*density));
    numberOfGrids.X = floor(numberOfGrids.X);
    numberOfGrids.Y = floor(numberOfGrids.Y);
    numberOfGrids -= 1.f;

    if (numberOfGrids.X<1||numberOfGrids.Y<1)
    {
        printf("GRASS GENERATOR ERROR: The density parameter is too large.\n");
        return;
    }
    //SColor for each pixel per plant, these are only testing arrays. In the end there is only going to be
    //a single array with a VT index
    video::SColorf* positionColours = new video::SColorf[u32(numberOfGrids.X*numberOfGrids.Y)];
    u8* positionColoursw = new u8[u32(numberOfGrids.X*numberOfGrids.Y)*4];

    //thread handles
    pthread_t threads[4];
    int rc;
    //whether the threads have been created and we can wait for them
    bool join=true;
    //input parameters for each thread
    imageMarchThreadInput td[4];
    //here we set thread specific boundaries
    //the +1 handles cases where the number of plants in X or Y direction is not exactly divisible by 2
    td[0].threadoffset = core::rectf(1,1,floorf(numberOfGrids.X/2),floorf(numberOfGrids.Y/2));
    td[1].threadoffset =
core::rectf(floorf(numberOfGrids.X/2)+1,1,numberOfGrids.X,floorf(numberOfGrids.Y/2));
    td[2].threadoffset =
core::rectf(1,floorf(numberOfGrids.Y/2)+1,floorf(numberOfGrids.X/2),numberOfGrids.Y);
    td[3].threadoffset =
core::rectf(floorf(numberOfGrids.X/2)+1,floorf(numberOfGrids.Y/2)+1,numberOfGrids.X,numberOfG
rids.Y);
    //creating the threads
    for(u32 i=0; i<4; i++){
        td[i].noOfGrids = numberOfGrids;
        td[i].posColours = positionColours;
        td[i].posColoursw = positionColoursw;
        td[i].vegMap = vegetationMap;
        rc = pthread_create(&threads[i], NULL, InterpolatePositions, (void *)(td+i));
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            join = false;
        }
    }
    //wait for threads to complete work
    for(u32 i=0; i<4&&join; i++)
```

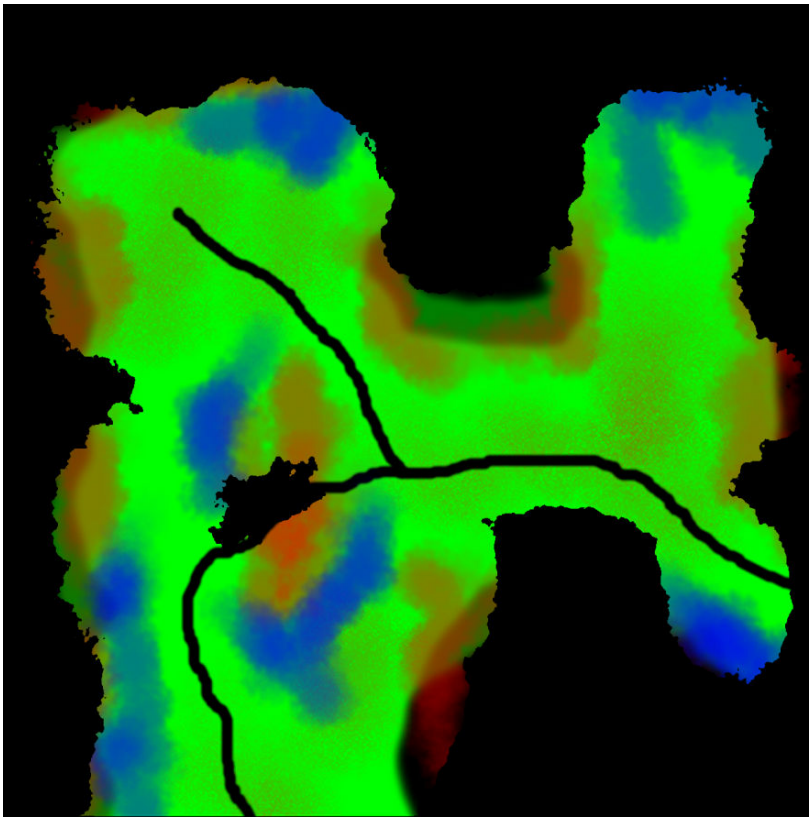
```
    (void) pthread_join(threads[i],NULL);

  //write the test array as an image to a file
  video->writeImageToFile(video->createImageFromData(video::ECF_A8R8G8B8,core::dimension2du(numberOfGrids.X,numberOfGrids.Y),positionColoursw),"./interpolated.bmp");

  //value to be used later on
  u32 no_of_plants = 0;

  printf("Possible number of plants: %f \n.",/*no_of_plants*/numberOfGrids.X*numberOfGrids.Y);
  //free the memory from the arrays
  delete [] positionColours;
  delete [] positionColoursw;

  return;
}
```

Next I decided to investigate whether multithreading gives me a performance boost. Using multithreading the image parsing took 610 milliseconds. Without multithreading, the same function took 1190 milliseconds. I am only showing the temporary modifications to the code I have made in order to measure the time taken and to carry out the process in a single thread (Bits in green are commented out temporarily).

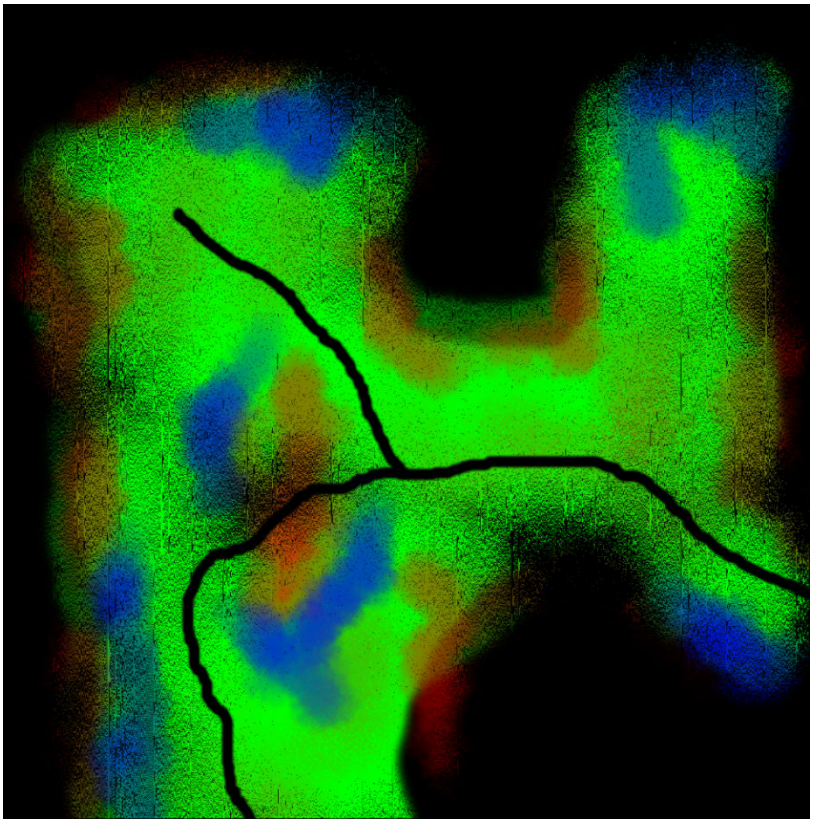
```
  u32 startTime = irrlicht->getTimer()->getRealTime();
  /*pthread_t threads[4];
  int rc;
  bool join=true;*/
  imageMarchThreadInput td[4];
  td[0].threadoffset = core::rectf(1,1,floorf(numberOfGrids.X/2),floorf(numberOfGrids.Y/2));
  td[1].threadoffset = core::rectf(floorf(numberOfGrids.X/2)+1,1,numberOfGrids.X,floorf(numberOfGrids.Y/2));
  td[2].threadoffset = core::rectf(1,floorf(numberOfGrids.Y/2)+1,floorf(numberOfGrids.X/2),numberOfGrids.Y);
  td[3].threadoffset = core::rectf(floorf(numberOfGrids.X/2)+1,floorf(numberOfGrids.Y/2)+1,numberOfGrids.X,numberOfGrids.Y);
  for(u32 i=0; i<4; i++){
    td[i].noOfGrids = numberOfGrids;
    td[i].posColours = positionColours;
    td[i].posColoursw = positionColoursw;
    td[i].vegMap = vegetationMap;
    InterpolatePositions(td+i);/*
    rc = pthread_create(&threads[i], NULL, InterpolatePositions, (void *)(td+i));
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      join = false;
    }*/
  }/*
  for(u32 i=0; i<4&&join; i++)
    (void) pthread_join(threads[i],NULL);*/
  printf("CALCULATION TOOK %u Miliseconds.\n",irrlicht->getTimer()->getRealTime()-startTime);
```

Lastly here is the alpha channel pruning visualized:

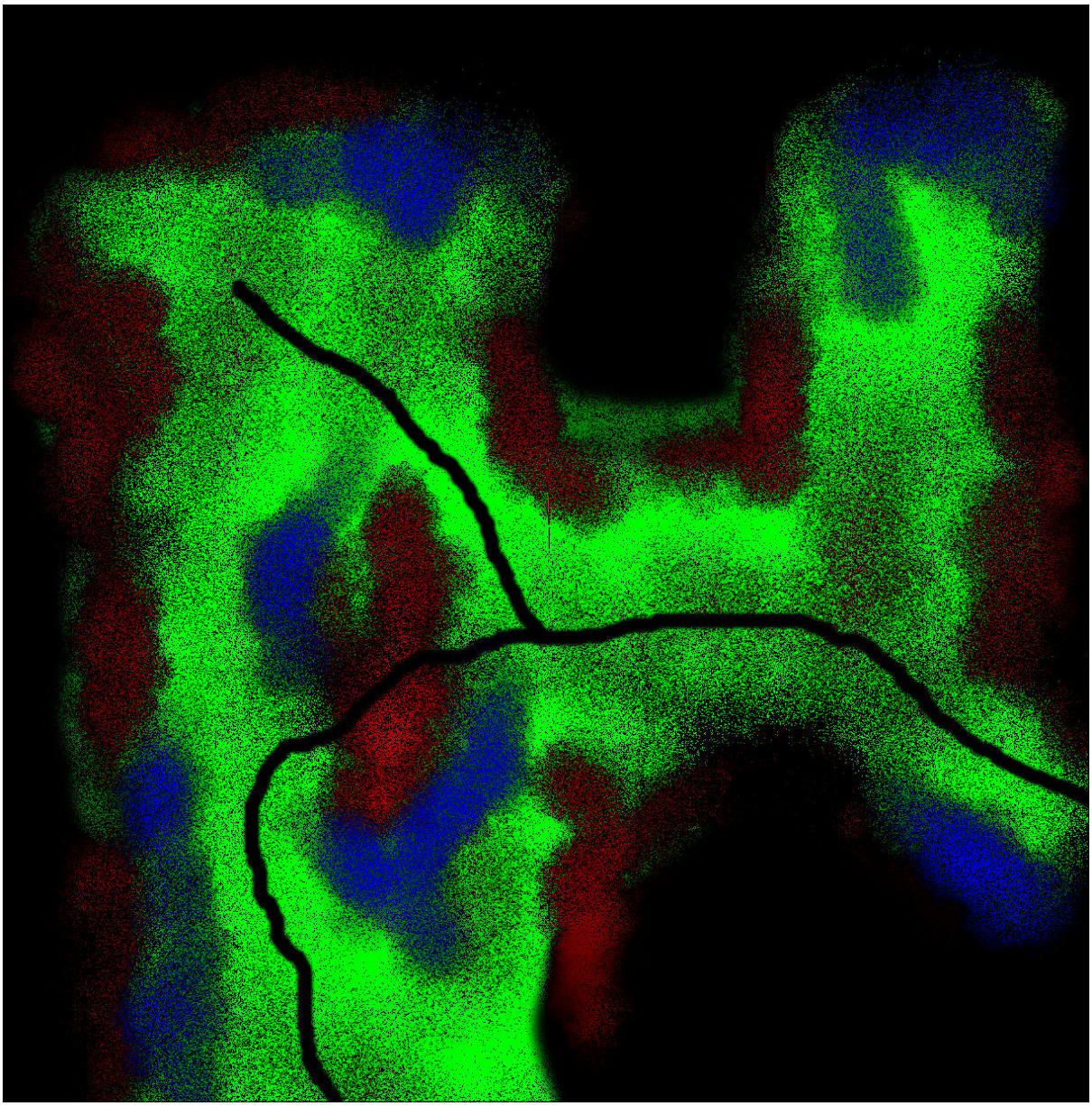
No random seed (alpha>127):



With random seed (alpha>127):



After all these changes I got rid of the temporary debug data and began the stratified colour channel choosing. Here is the result of the processing after the application chooses which colour channel vegetation type will be planted.



each pixel represents a plant