What are TinkerCell extensions and How to write them

Introduction

TinkerCellCore (or just "Core") is a C++ library for creating network drawing programs. The Core library supplies numerous operations such as copy/ paste, undo/redo, user interface, etc. Extensions written in C++ are classes that extend the Tinkercell::Tool class and add new functions or interfaces on top of the Core library. The extensions have lots of freedom. Using extensions, it is possible to completely change the way the user interacts, change the windows, buttons, add new windows, and so on. The C++ extensions can also add new functions to the C application programming

interface (API). The C API automatically generates a Python API, Octave API, and Ruby API. Programs written in C, Python, Octave, or Ruby are called "plug-ins". They are fundamentally different from extensions because they have much less freedom. Plug-ins are easy to write and share between TinkerCell users, but extensions serve as a means of creating more significant changes to TinkerCell.

This document will explain extensions via examples. All the examples are available with the TinkerCell source code in the ExamplesUsingCore folder. In the ExamplesUsingCore folder, there are two small programs, MultiCell and SimpleDesigner, that are good demonstrations of using TinkerCell Core library to create a GUI application.

After reading the examples, it would be best to take a look at the code documentation to see the list of available classes and functions, especially in MainWindow, NetworkHandle, and GraphicsScene.

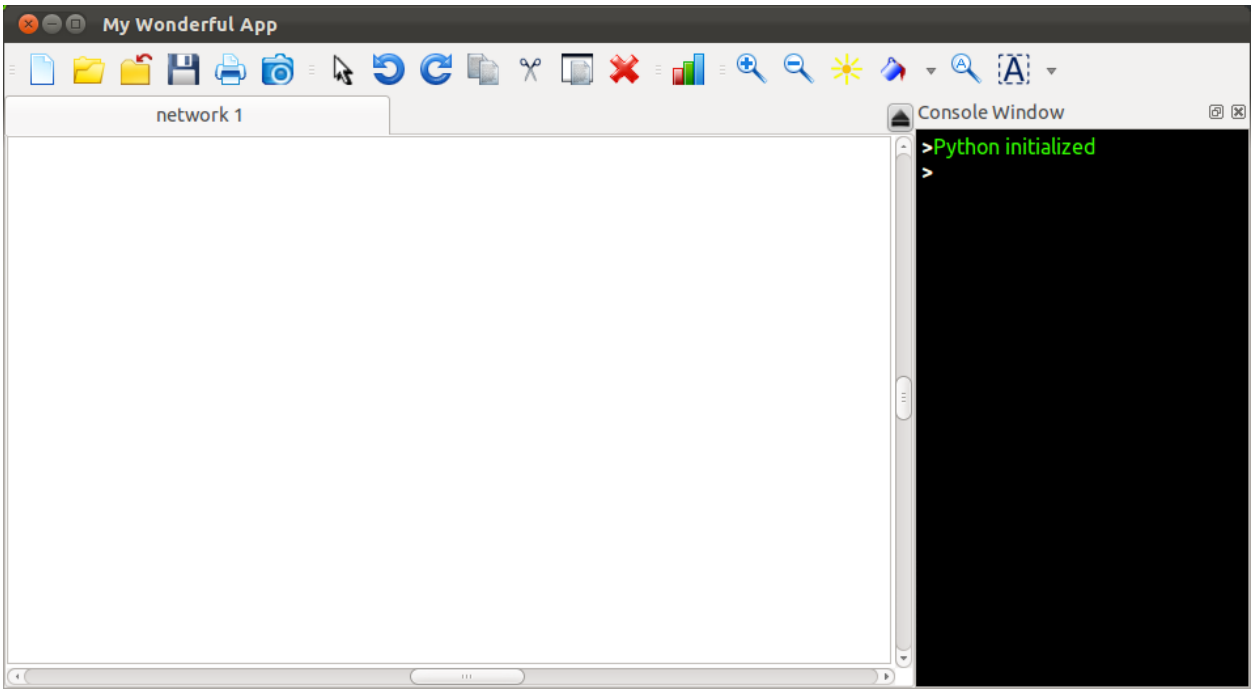
Main()

Remember than TinkerCellCore is a library, not the actual program itself. The actual program using TinkerCellCore would contain the main() function, which would create the Tinkercell::MainWindow and setup some other optional variables. Below is an example code. Note the comments "required", "recommended", and "optional". The rest of the plug-ins discussed in this document will assume that this main function exists.

```
#include "MainWindow.h"
#include "GlobalSettings.h"
#include "Ontology.h"

using namespace Tinkercell;

#if defined(Q_WS_WIN) && !defined(MINGW)
```

```
int WinMain(int argc, char *argv[])
#else
int main(int argc, char *argv[])
#endif
{
    //setup project name -- REQUIRED
    GlobalSettings::PROJECTWEBSITE = "www.tinkercell.com";
    GlobalSettings::ORGANIZATIONNAME = "My Wonderful Co.";
    GlobalSettings::PROJECTNAME = "My Wonderful App";

    //start Qt -- REQUIRED
    QApplication app(argc, argv);
    QString appDir = QCoreApplication::applicationDirPath();

    //enable of disable features -- OPTIONAL (see default values in
GlobalSettings.h)
    GlobalSettings::ENABLE_HISTORY_WINDOW = true;
    GlobalSettings::ENABLE_CONSOLE_WINDOW = true;
    GlobalSettings::ENABLE_GRAPHING_TOOLS = true;
    GlobalSettings::ENABLE_CODING_TOOLS = false;
    GlobalSettings::ENABLE_PYTHON = true;
    GlobalSettings::ENABLE_OCTAVE = false;
    GlobalSettings::ENABLE_LOADSAVE_TOOL = true;

    //create main window -- REQUIRED
    MainWindow mainWindow(true,false,false);  //@args: enable scene, text,
allow pop-out windows
    mainWindow.readSettings();   //load settings such as window positions

    //set window title -- OPTIONAL
    mainWindow.setWindowTitle("Simple Designer");

    //ADD PLUGINS

    //load Ontology -- OPTIONAL
```

```
    Ontology::readNodes("Nodes.nt");
    Ontology::readConnections("Connections.nt");

    //create an empty canvas -- RECOMMENDED
    GraphicsScene * scene = mainWindow.newScene();
    mainWindow.show();
    int output = app.exec();

    return output;
}
```



How to Build Extensions

Since TinkerCell is built using the Qt framework, the build procedure is the same as any Qt project. The main program given above can be built manually using the following command in Linux:

```
g++ simplemain.cpp -ICore -LBUILD/bin -lTinkerCellCore -I. -I/
usr/include/qt4/QtGui/ -I/usr/include/qt4/QtCore/ -I/usr/include/
qt4/QtXml -I/usr/include/qt4/QtOpenGL/ -I/usr/include/qt4/ -o
BUILD/bin/simplemain.out
```

Then to run:

```
cd BUILD/bin
export LD_LIBRARY_PATH=.
./simplemain
```

However, the simpler option is to use qmake or CMake. Since the TinkerCell project is organized using CMake, the best option is probably CMake. In the TinkerCell source, the ExamplesUsingCore/Sample_CPP_Extensions/ CMakeLists.txt shows the CMake commands for building the above program:

```
add_executable( simplemain simplemain.cpp)
target_link_libraries( simplemain TinkerCellCore )
```

The above program does not have any header files. Header files require a bit more processing. Specifically, there is a program called MOC that is used by Qt to perform some preprocessing. MOC needs to run before compilation. In CMake, this is done by the qt4_wrap_cpp command. Here is an example where SamplePlugin1.h and SamplePlugin1.cpp are also included in the program:

```
qt4_wrap_cpp( testPlugin1hrs SamplePlugin1.h )
add_executable( testPlugin1 SamplePlugin1.cpp simplemain.cpp
${testPlugin1hrs} )
target_link_libraries( testPlugin1 TinkerCellCore )
```

Note that the the include directories, link directories, and some other definitions have been added in the CMakeLists.txt file in the trunk/ folder, so

the above CMake commands assume that those things have already been performed.

How to Load an Extension

There are two ways to load extensions: statically or dynamically.

Static loading just means that the extension class is added to the main window using the addTool method, e.g. mainWindow.*addTool*(new SamplePlugin1)

Dynamically loading an extension requires (1) creating a dynamic library and (2) loading the dynamic library by either calling mainWindow.*loadDynamicLibrary* or mainWindow.*loadFiles*. Node that when a dynamic library is dropped on a canvas in TinkerCell, mainWindow.loadFiles gets called automatically, so dragging the library onto the TinkerCell window is another option.

If creating dynamic libraries for extensions, then the dynamic library MUST contain the following function, with MyTool replaced the extension class:

```
	extern "C" TINKERCELLCOREEXPORT void
loadTCTool(Tinkercell::MainWindow * main)
	{
		if (!main) return;
		Tinkercell::MyTool * myTool = new
Tinkercell::MyTool;
		main->addTool(MyTool);
	}
```

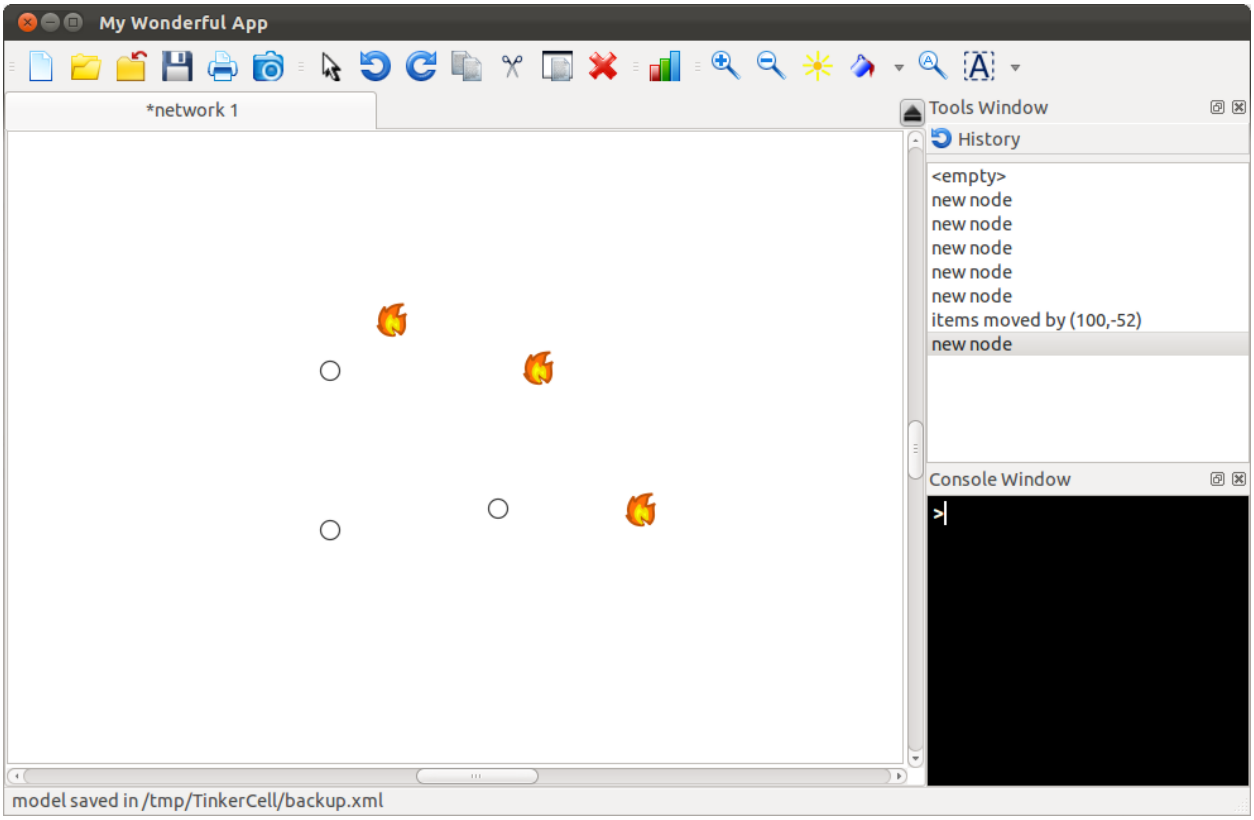
Dynamic loading is useful if you want to create a directory of third-party plugins that get loaded automatically (i.e. write a loop for getting all files in a folder using QDir).

The examples in this document will use static loading, since it is simpler.

Example 1: Simple C++ Extension

This example will create an application where the user can create simple nodes by clicking on the screen and create "fire" nodes by holding ctrl while clicking the mouse button. The TinkerCell Core library will automatically provide undo/redo, selecting, moving, naming, and copy/paste features. In addition the Core library will provide the optional Python console. The Python console in this example will have all the functions listed in TC_Main_api.h

To build this program using the existing TinkerCell code, just do "make testPlugin1" in the Build folder (follow TinkerCell installation guidelines).



Header file (SamplePlugin1.h):

```
#include "Tool.h"

namespace Tinkercell
{

   class SamplePlugin1 : public Tool
   {

      Q_OBJECT

   public:
      SamplePlugin1();
      bool setMainWindow(MainWindow * main);

   public slots: //signals and slots work like events
      void mouseReleased(GraphicsScene * scene, QPointF point,
Qt::MouseButton button, Qt::KeyboardModifiers modifiers);
      void itemsSelected(GraphicsScene * scene, const
QList<QGraphicsItem*>& items, QPointF point, Qt::KeyboardModifiers
modifiers);
   };

}
```

Source file (SamplePlugin1.cpp):

```
#include "MainWindow.h"
#include "NetworkHandle.h"
#include "NodeGraphicsItem.h"
#include "TextGraphicsItem.h"
#include "ItemHandle.h"
```

```
#include "SamplePlugin1.h"
using namespace Tinkercell;

SamplePlugin1::SamplePlugin1(): Tool("My Plugin 1", "Sample Plugins")
//name, category
{
}

bool SamplePlugin1::setMainWindow(MainWindow * main)
{
   Tool::setMainWindow(main); //must call this to properly setup the extension

   //this extension performs some action when mouse button is released
   connect(main, SIGNAL(mouseReleased(GraphicsScene*, QPointF,
Qt::MouseButton, Qt::KeyboardModifiers)),
               this, SLOT(mouseReleased(GraphicsScene*, QPointF,
Qt::MouseButton, Qt::KeyboardModifiers)));

   //this extension performs some action when items on the screen are selected
   connect(main, SIGNAL(itemsSelected(GraphicsScene *, const
QList<QGraphicsItem*>&, QPointF, Qt::KeyboardModifiers)),
               this,SLOT(itemsSelected(GraphicsScene *, const
QList<QGraphicsItem*>&, QPointF, Qt::KeyboardModifiers)));
}

void SamplePlugin1::mouseReleased(GraphicsScene * scene, QPointF point,
Qt::MouseButton button, Qt::KeyboardModifiers modifiers)
{
   /*
      One network can be represented in multiple scenes (canvas) or
      multiple text editors, so a scene might be just a piece of the entire
network
   */
   NetworkHandle * network = scene->network;  //get the network
```

```
    NodeHandle * handle = new NodeHandle("x"); //create a new "Handle", or
entity, called x
    scene->network->makeUnique(handle);  //assign unique name, if x is
already taken

    NodeGraphicsItem * node;  //greate a graphical node, different from a
Handle
    if (modifiers == Qt::ControlModifier)
        node = new NodeGraphicsItem(":/images/fire.xml");  //load either the fire
image
    else
        node = new NodeGraphicsItem(":/images/defaultnode.xml"); //or the
default image

    node->setHandle(handle);  //set the handle for the graphical item
    TextGraphicsItem * text = new TextGraphicsItem(handle);  //create a text
box that also belong with the same handle

    node->setPos(point); //set the position of the node. point is where mouse
was clicked

    QPointF bottom(0, node->boundingRect().height()/2);  //set the position of
the text box
    text->setPos( point + bottom );
    QFont font = text->font();  //increase the font of the text box
    font.setPointSize(22);
    text->setFont(font);

    QList<QGraphicsItem*> list;  //create a list of the node and the text box
    list << node << text;

    scene->insert("new node", list);    /*insert items into the scene.
    DO NOT USE scene->addItem because that is part of the Qt
    and would not do all the things that ->insert would do*/
}
```

```
void SamplePlugin1::itemsSelected(GraphicsScene * scene, const
QList<QGraphicsItem*>& items, QPointF point, Qt::KeyboardModifiers
modifiers)
{
   /*
      The following look ensures that text boxes also move
       when nodes are selected, but nodes do not move when
      text boxes are selected

      A handle is a collection of graphical items that reprent
      the same entity. For example a node and the text box
      below the node belong with the same entity.
   */
   for (int i=0; i < items.size(); ++i)
   {
      ItemHandle * h = getHandle(items[i]);
      if (h != 0 && NodeGraphicsItem::cast(items[i]))
         scene->moving() += h->graphicsItems;
   }
}

/*
copy main() from the earlier example here and add the line
mainWindow.addTool(new SamplePlugin1);
*/
```

Example 2: Overriding the Default Behaviors

coming ...
Basically, this involves setting GraphicsScene::useDefaultBehavior(false) and connecting to all

the mouse and keyboard signals.
ExamplesUsingCore/MultiCell is a good example of this.

Example 3: Extending the C, Python, Octave, Ruby API

coming ...
The ImportExportTools/SBMLImportExport.h and .cpp are good examples of this.

Example 4: Interacting with Other Extensions

coming ...
Basically, this involves three steps:
1) include the header file from the relevant extension, e.g. #include "PlotTool.h"
2) in setMainWindow, find the extension using mainWindow->tool("name of tool")
The easiest way to get extension names is by using the "Packages" menu in TinkerCell which
lists all the extensions that have been loaded.
3) connect to the signals or slots in that extension
Note that one tool can prevent another tool from loading via the toolAboutToLoad signal in MainWindow