CS371 Tools Overview

Updated September 3, 2010

In CS371 you'll use a development environment similar to what you would encounter in professional development. It comprises a C++ build system, revision control, a debugger, documentation tools, profiling tools, and many software libraries. Most industry developers use commercial, visual integrated development environments (IDEs) like Visual Studio. In class we favor command-line open source tools . Learning these tools may help you understand the fundamentals better than the visual environments. What you learn with these tools is directly applicable to the visual environments, and they are always available to you for future courses and work environments because they are that are cross-platform and freely available.

This document briefly introduces the software development environment for CS371. It is intended as an introduction and quick reference guide. Refer to the online manuals and guides [van Heesch 2010; Roberts 2009; Collins-Sussman et al. 2008; McGuire 2010], OS X man pages, and built-in help commands for more detailed information. I'm intentionally telling how to find information rather than giving you the information directly so that you will learn to work with reference materials and external resources.

Contents

1 Subversion

1.1 Revision Control

Subversion is a **revision control system**. Revision control maintains a server-side **repository** (i.e. database) of the files in your project. You can **check out** (i.e., download) a copy of these files to your local machine, into what is often called a **workspace**. You then develop with the local copy and **commit** your changes back to the server, which merges your changes into the files already there. Commits usually occur at the end of your programming session or after completing some milestone. Because commits merge files, you can modify your program on multiple computers and you individual changes will be integrated at the server. Multiple programmers can also modify files from the project simultaneously and independently, and then rely on the merge to integrate them. Once you have a workspace, you can also **update** it by merging any changes from the server side made since check out time into your workspace. Most software today, both in research and industry, is developed using revision control to manage project files. That is because of the many advantages it offers, including:

1. History–you can jump back to the state your project had at any previous commit point. This is particularly useful if some new change introduced a bug or accidentally removed a component.

2. Asynchronous development–multiple developers can work on the same code base without tightly coordinating.

3. Multi-computer development–you can use the fast local disk for work and rely on revision control for moving files between computers, rather than explicit copying which is prone to error.

Revision control has drawbacks as well. To avoid these, adopt the following practices:

1. Always add new files to the system as soon you create them. Adding does not commit.

2. Always commit before leaving a machine, and then update to see if you forgot to add new files.

3. Update and build every time you sit down at a computer. This will alert you if the build is broken before you make new changes.

4. Always run `svn status` in your project root before you log out to make sure you checked everything important in.

5. Work in small increments, committing frequently.

6. Only commit working builds. Use `if (false)` or comments to temporarily disable broken code if you have to end your programming session at a specific time.

7. Avoid editing the same files, and especially the same methods, simultaneously with your partner. The system cannot merge changes to the same line of code and changes within the same method are likely to merge but risk incorrect semantics.

8. Never copy or move directories that are under revision control.

9. Never modify the `.svn` directories.

10. Never add generated files (e.g., executables, generated documentation) to the repository.

11. Avoid adding large binary files (e.g., 20 MB movies, PSD files), and especially avoid changing such large binary files because Subversion cannot merge these, so they consume tremendous server space and slow down the system.

1.2 Commands

You will access Subversion through the `svn` command-line program. Issue subversion commands by running `svn` with arguments specifying the operation you would like to perform and any options that command requires. The major commands that you will use are:

`svn co` *source-URL*

`svn update`

`svn add` *filename*

`svn commit -m "` *log message* `"`

`svn export [--force]` *source-URL* *dest-dir*

`svn status`

To tell Subversion to ignore a file, use:

`svn propset svn:ignore` *file-pattern* *containing-dir*

For, example,

```
svn propset svn:ignore log.txt data-files
```

Refer to the Subversion manual [Collins-Sussman et al. 2008] or use the `svn help` command for other useful commands and for the details of these.

When you commit you must specify a log message. Make this a one sentence description of your changes. These will help you if you need to revert a change and will help your partner (in future projects) to understand what new code has come in with an update.

1.3 Starting Each Week

For each project I will create a Subversion module for you. This will either have your username or an assigned team name in the directory name.

Your workspace will initially be an empty directory. For most projects, you'll quickly fill this by copying your solution (or another student's) from the previous week. You can't just copy the directory structure of another project directly because Subversion maintains its state in subdirectories named `.svn`. If you copy a `.svn` subdirectory, you will corrupt the state of your working copy. Copying would also bring along generated files like executables that you don't want.

Use the Subversion `export` operation to export a previous solution from the server and strip its revision control data. You can then check this back in as a different project. If your username was `ewilliams` and you were working on Project 1, the commands for this process would be:

```
cd /local-scratch
svn co svn://graphics-svn.cs.williams.edu/371/1-Meshes/ewilliams-meshes ewilliams-meshes
svn export --force svn://graphics-svn.cs.williams.edu/371/0-Cubes/ewilliams-cubes ewilliams-meshes
cd ewilliams-meshes
svn add *
svn commit -m "Exported from previous week"
```

2 iCompile

iCompile is an automated build system for C++ on Linux and OS X. It provides similar functionality to tools like Make, MSBuild, and Ant. What makes iCompile unique is that it generally requires no configuration. You just run `icompile` with no options in the root directory of your project and it automatically determines dependencies, directories, and compiler and linker options and builds your program. You can also use it to build documentation, shared and static libraries, and standalone OS X distributions (.dmg files).

iCompile is implemented as an open source Python script that is installed as part of the G3D distribution. Run `icompile --help` to see a full list of options. Some of the most common are:

`--run [... args ...]` If compilation succeeds, run the program. Arguments can be passed on after the run flag.

`--gdb [... args ...]` If compilation succeeds, run the program under a debugger.

`--clean` Delete all generated files.

`--doc` Generate documentation from Doxygen markup.

`--opt` Build an optimized executable.

You can customize iCompile's behavior by editing `~/.icompile` and the project's `ice.txt` file.

2.1 Directory Organization

iCompile can work with almost any directory structure. However, it treats certain directory names specially to support common development needs. For CS371, I want you to take advantage of this by structuring all of your projects with the following subdirectories:

`source`	All of your source code, divided among `.h` and `.cpp` files, and the source for your report and overview documentation in `.dox` files.
`data-files`	Any runtime data required by your program that is *not* also in the G3D data distribution.
`doc-files`	Any data required for your report, such as images and videos. Do *not* put the `.dox` files here.
`graveyard`	Files that you want to keep around for your own reference but do not want me to evaluate or the build scripts to process.

You must use the exact naming scheme described here, including capitalization, to ensure that the scripts I use to process the projects work correctly. The naming scheme is part of the specification for each project and you will lose points for varying from it!

3 Coordinate System

Every 3D system imposes its own coordinate system conventions. These are arbitrary–everything that you'll learn in this course works equally well in any coordinate system, and it is straightforward to convert between them.

3.1 3D

In the 3D coordinate system used in this course, the x-axis increases to the East, the y-axis increases vertically upwards, and the z-axis increases to the South.

This is a **right handed** coordinate system. If you point your right hand in the direction of the x-axis and curl your fingers towards the y-axis (which necessitates having your palm upwards), then your thumb will be pointing along the z-axis. This works for any cyclic rotation of the order of axes, e.g., x-y-z has the same relationship as y-z-x.

3.2 World and Object Space

We distinguish between the absolute **world-space** (a.k.a. global) coordinates in which we will define the entire world (a.k.a. **scene**) and the relative **object-space** (a.k.a. body-space, local) coordinates used to define parts of an object relative to the reference frame of that object. For example, I might position a chessboard relative to the center of the scene, but the pieces on the board relative to the board itself.

By convention we will generally define object space coordinate systems in a common way. For objects that have a clear "top," we will make their object space y-axis point upward. For objects that have a natural "facing" direction, such as cars and people, we will define their object space z-axis to point out their back and the x-axis to point to their right. Thus objects look along their negative z-axis.

3.3 Rotations

The canonical rotations about the x-, y- and z-axes are called **pitch**, **yaw**, and **roll**. These also follow a right hand rule: if you point your thumb in the direction of the axis of rotation, your fingers curl in the direction of increasing angular measure.

3.4 2D

In the 2D coordinate system used in this course for images and the screen, the origin is at the upper-left corner. The x-axis increases to the right and the y-axis increases downward. The reading discusses the historical origin if this coordinate system.

Image space coordinates are sometimes expressed in pixel side-lengths, e.g., position (100, 120) on a 1920×1080 image. At other times they are in normalized so-called **texture coordinates**, in which (1, 1) is the lower-right corner of the image regardless of its resolution or aspect ratio. Texture coordinates are often expressed using the variables (u, v) or (s, t) to distinguish them.

3.5 Units

We use SI units (e.g., meters, seconds, Joules, Watts), which include radians as the unit of planar angle measure.

4 The C++ Memory Model

This section briefly overviews a subset of the ways of working with memory and types in C++. You need to know about these features, especially pointers, because you will interact with APIs that require you to use them and you will design your own data structures that require them. Beware that pointers in C++ are a very dangerous feature that are responsible for many of the crashes observed in commercial software, and that you can almost avoid using them.

If you're used to programming in Java, you need to get out of the habit of writing `new` every time you create a value. You also need to always be conscious of: how large your values are, whether values are in the stack or the heap, and when you are referring to a pointer to a value vs. a value itself.

You can avoid most memory management related program errors by following these practices:

- Allocate small objects on the stack whenever possible
- Allocate only one variable per declaration (e.g., `int x; int y` instead of `int x, y`)
- Always initialize variables (C++ does not define the value of uninitialized variables!)
- Avoid using explicit pointers–favor references and reference-counted pointers
- Avoid ever using `new` except in static factory methods of reference-counted objects
- Avoid the address-of operator and pointer arithmetic
- Use C++ strings (`std::string s`) instead of C-strings (`char* s`)
- Use C++ arrays (`G3D::Array<T> a` or `std::vector<T> a`) instead of C-arrays (`T a[]`)

4.1 Types

A **variable** is a name in a programming language that refers to a **value**. That value is stored at some **address** in memory. The **type** of the value specifies the interpretation of the bytes in memory at the address. C++ is a statically-typed language, meaning that the type of a variable is determined at compile time and never changes (the type of a *value* of course never changes in any language).

A variable is declared by the type followed by the name, for example,

```
int x;
App app;
```

Some examples of types in C++ are `int`, `std::string`, and `float`. You can create new types are created using class declarations.

4.2 Pointer Types

Indirection is a useful tool in programming. Sometimes we don't want to refer directly to a value, but to the location at which the value is stored. In this case we use a **pointer** type. In Java, all variables except those with primitive types are implicitly pointer types. In C++, pointer types are explicit. A pointer type consists of the type of value that is being pointed at, followed by an asterisk. For example, `int*` is the type of a pointer to an `int`. You can make a pointer to any value. You can also make pointers to pointers. The following are all legal declarations of pointer variables:

```
int* x;
float** y;

class Foo {
   int z;
};

Foo* f;
```

Pointers are only useful when they point at something. The value of a pointer is an address. There are three common ways of obtaining an address: calling a function that returns a pointer, allocating memory, and using the address-of operator. The `new` operator is the safest way of allocating memory. It allocates a block of memory on the heap, invokes the constructor of the specified type, and then returns a pointer to that new object. Memory allocated by `new` can later be deallocated by `delete`, which invokes the destructor on the object and frees the memory. It is a good idea to set the pointer variable to the `NULL` address afterward to avoid accidentally referring to the deallocated memory block. For example,

```
int* x = new int();
...
delete x;
x = NULL;
```

The address-of operator returns the address at which a value is stored, for example:

```
int y = 3;
int* x = &y;
```

It is somewhat dangerous to use the address-of operator because one must be careful to ensure that the pointer is only used when the referenced memory is still allocated.

A pointer value cannot be used directly. Instead, one must **dereference** the pointer using the dereference operator to obtain the value that was pointed at. The dereference operator is an asterisk that is placed before the pointer value. For example:

```
float* q = new float(3.14f);

// Operate on the value pointed at by q
*q = *q * 2.7f;

delete q;
q = NULL;
```

When the value pointed at has member variables or methods, these can be accessed more concisely with the arrow operator.

```
std::string* s = new std::string("hello");

// This is the same as:  (*s).substr(3);
s->substr(3);

delete s;
s = NULL
```

4.3 Stack and Heap Allocation

All variables declared within a function or method refer to memory on the stack. Member variables of classes refer to memory in the same location as the class instance.

Variables are declared within scope lines demarcated by curly braces. This means that the memory associated with a variable is no longer assigned to that variable when the program exits the scope line. Variables on the stack do not have to be explicitly allocated–the declaration implies the allocation. They also do not have to be explicitly deallocated.

The address-of operator allows you to obtain a pointer to memory on the stack. Nothing prevents you from retaining that pointer after the program exits the scope line. Pointers to values from variables that are no longer in scope are called **dangling pointers**, meaning that they reference memory that is no longer allocated for its original purpose. Dereferencing a dangling pointer will read or write an untended value, which will corrupt memory and possibly crash your program. This is one reason that the the address-of operator is so dangerous: it can be used to obtain the address of a value on the stack that may then go out of scope.

To create a value that persists beyond a scope line, allocated typed, constructed memory in the heap with `new` or untyped, uninitalized memory in the heap with `malloc`. G3D also provides some optimized variations on `malloc`– see `G3D::MemoryManager` and its subclasses.

A **memory leak** occurs if you forget to later free the memory that was allocated on the heap. Leaks do not corrupt memory, although you could run out of memory at some point. Nothing prevents you from retaining a pointer to memory that you have explicitly freed, so you can create dangling pointers by freeing memory and then using pointers to it anyway. Nothing prevents you from freeing the same memory twice, which will throw an exception, crash your program, or corrupt memory. Since you may have more than one pointer to the same block of memory, it can become tricky to know when it is safe to free.

Hopefully all of these potential errors are convincing you that manual heap memory management is hard and dangerous, and that you should stack allocate almost everything and rely on well-tested data structures (like `G3D::Array`) to manage the heap for you.

Note that an object that references a large amount of data may itself consume a small space on the stack. For example, a `G3D::Array` containing one million 32-bit integers takes about 8 bytes of stack space–it only stores the length of the array and a pointer to some memory in the heap. Thus allocating the array object on the stack is very reasonable...but beware that assigning one array to another will trigger a giant copy.

4.4 References

A C++ **reference** type has the syntax of a value and the semantics of a pointer. It allows you to create a new name for a value that is already in memory in a relatively safe way. A reference type has an ampersand after the type of the referenced value. References must be initialized at creation, which means that they are never `NULL`.

For example,

```
// A value on the stack
int x = 7;

// A reference to the same value as x
int& y = x;

y = 3;
```

```
// Both x and y are now 3!
```

Since assignment overwrites a value with a new value, it can copy significant amounts of memory. Copying is slow, and often not the semantics we want anyway. Therefore references and `const` (immutable) references are frequently used for passing parameters to functions and methods. References are rarely used as return values because the return value of a function is frequently an intermediate value or a local variable–both of which are on the stack and will be undefined when the function returns.

4.5 Reference Counted Pointers

A **reference counted pointer** has syntax and semantics like that of a pointer, except that it manages the referenced memory automatically. Each reference counted object has an internal count of the number of pointers that *reference* it. When that count reaches zero, there is no way of referring to the object, so the object knows that it can safely delete itself. This avoids dangling pointers and provides the garbage collection semantics of languages like Java and Python. This does not prevent memory leaks because a cycle of pointers (e.g., an object with a reference counted pointer to itself) will keep all of the counts along the cycle above zero even though there are no external pointers into that cycle.

C++ has no built-in reference counted pointer type, but the language is rich enough that libraries can provide them. The standard library defines `auto_ptr` and `shared_ptr`, which have reference-counting-like semantics. These don't do exactly what you would expect, however, so you should use G3D's `G3D::ReferenceCountedPointer`.

To use an existing reference-counted type you write code like:

```
// Acts like Image*, but you don't have to free the
// memory yourself
Image3::Ref image;

// All reference-counted objects provide a static factory
// method; don't use new
image = Image3::createEmpty(128, 32);

// Invoke methods just as if with a raw C pointer
image->set(10, 10, Color3::blue());

// Create aliased pointers
Image3::Ref b = image;

// Whenever the last pointer to this object is set to NULL
// (or the pointer variable goes out of scope)
// the image will be destroyed and the memory reclaimed.
b = NULL;
image = NULL;
```

To create your own reference-counted class, use the following pattern:

```
// header:
class Foo : public ReferenceCountedObject {
public:
    // Makes a shorthand Foo::Ref instead of forcing the programmer
```

```
    // to write out ReferenceCountedPointer<Foo>
    typedef ReferenceCountedPointer<Foo> Ref;

private:
    ...

    // Make your constructors private to prevent programmers
    // from invoking "new Foo()" anywhere.
    Foo();
    ...

public:

    // Define a factory method
    static Ref create();
    ...
};

//////////////////////////////////////////////////////////
// source file:
Foo::Ref Foo::create() {
    return new Foo();
}
```

4.6 Copying and Assignment

C++ provides many ways of copying and initializing variables. This is a weakness of the language and you should use a few idiomatic forms to avoid confusion. Always initialize variables in their declarations using parentheses (think of this as invoking the constructor) instead of using the assignment operator:

```
// Good: Create y on the stack with an initial value of 3.
int y(3);

// Bad: Create x on the stack uninitialized, create a temporary
// value 3 on the stack, copy 3 to x, and then release the
// temporary value.
int x = 3;
```

In the case of a single integer, the compiler will actually generate the same code in both cases. But if you were creating a larger object, the second case would be inefficient. For example:

```
// Good: creates one empty array
Array<int> y;

// BAD: Creates two empty arrays and copies one over the other
Array<int> y = Array<int>();

// Illegal: assigning a pointer to a non-pointer variable
Array<int> y = new Array<int>();

// Bad: you probably want an array, not a pointer to an array
Array<int>* y = new Array<int>();
```

```
// Bad and illegal: Now you're just writing random pieces
// of code and hoping it will work.  Come to office hours.
Array<int>::Ref z = new Array<int>::Ref()
```

4.7 Pre- and Post-Increment

Get in the habit of writing `++i` instead of `i++`. Here's why.

C++ uses incrementable iterator objects to concisely iterate through the elements of data structures, e.g.,

```
Table<std::string, int> table;
...
for (Table<std::string, int> it = table.begin(); it.hasMore(); ++it) {
   printf("%s %d\n", it->key.c_str(), it->value);
  ...
}
```

By convention, the pre-increment operator in the `for`-loop means that the iterator object should move to the next element of the table. That iterator object may have a substantial amount of state inside it. If we post-increment (`it++`), the compiler is forced to make a copy of the iterator object to return as the value of the post-increment operation. That copy could be very expensive. If we pre-increment (`++it`), then the compiler need not make a copy. This makes no difference in languages like C or Java that only support increment on primitive types, but is tremendously important in C++.

5 Doxygen

Write entry point (class, method, function, macro, enum, typedef) documentation and final report as **Doxygen** comments inside your C++ header (`.h`) files and standalone `.dox` files, all stored in the `source` directory.

Put images and other files referenced from your documentation in the `doc-files` subdirectory. iCompile will copy them when you build documentation.

Like HTML and LaTeX, Doxygen is a markup language that you use to *edit* a document. To actually *view* the document, you must compile it. To compile the document, execute the command `doxygen` with no arguments in the directory containing a file named `Doxyfile`. The Doxyfile that you will use for all projects is provided on the course webpage. You never need to modify it, although you may if you wish.

The following is a brief overview of some of the features of Doxygen. Read the manual [van Heesch 2010] for full details. **Sharing markup tips and helping classmates with formatting is one way to earn class participation, so please collaborate on this and let me know at the end of your report if you gave or received assistance.**

5.1 Markup

Doxygen comments begin with `/**` and end with `*/`. They apply to the entry point immediately following the comment. Only the markup in your header files and in .dox will affect your generated documentation.

An example of how to document a class is:

```
/** Represents a direction and magnitude in 3D. */
class Vector3 {
public:
    /** Distance along the x-axis. */
    float x;

    ...

    /** Magnitude of the vector. */
    float length() const;
};
```

You may have exactly one `\mainpage` markup command throughout your program. This declares that the containing comment forms the `index.html` page that will be your report. Put this in a .dox file in the source directory, e.g.,

```
/**
\mainpage

<b>Project 0: Cubes</b>
<br>Ephram Williams

\section outline Code Outline
App::onInit loads the scene...

\htmlonly
<center><img src="architecture.png" width=100></center>
\endhtmlonly
```

```
...

*/
```

5.2 Style

Doxygen markup commands begin with a backslash. Some useful ones are `\sa`, `\brief`, `\param`, `\author`, and `\return`. Doxygen also allows creation of nested lists using leading dashes and hash marks, and some HTML commands work as well. You can escape to raw HTML by creating a `\htmlonly`...`\endhtmlonly` block. See the manual for more markup commands.

5.3 Links

Doxygen will automatically hyperlink URLs and the names of entry points (e.g., methods, functions, classes, and variables) in your project. Make sure to check these links in your report–mispellings and incorrect capitalization can break them. Compare the G3D header source code and the generated documentation for a page, and remember that you can mine the G3D source for examples of how to achieve specific effects.

5.4 Equations

Within Doxygen comments, you can format standalone equations using LaTex markup inside blocks bracketed by `\f[` and `\f]`. For inline equations, use `\f$` to both begin and end the block. As an example, the following Doxygen source:

```
\f[ \int_{0}^{2\pi} \!\! \int_{0}^{\pi/2} \cos \theta ~ d\theta ~ d\phi = \pi \f]
```

Embeds this equation in your document:

$$\int_0^{2\pi} \int_0^{\pi/2} \cos\theta \; d\theta \; d\phi = \pi$$

I recommend Andrew Roberts' LaTex math tutorial [Roberts 2009] if you are unfamiliar with LaTeX.

If your LaTeX code contains an error, Doxygen may cache the erroneous result, which makes it hard to debug. When you suspect that this is happening, use `icompile --clean` to clear the cache.

6 G3D

The **G3D Innovation Engine** is an open source C++ library for 3D graphics on Windows, Linux, and OS X. It is used in commercial games, research papers, military simulators, and university courses. G3D supports hardware accelerated real-time rendering using OpenGL, off-line rendering like ray tracing, and general purpose computation on GPUs.

No 3D developer programs directly on the C++ standard library and OpenGL or DirectX. They are at too low of a level and don't provide necessary facilities such as scene management, image I/O, GUIs, and platform abstraction. Instead, programmers adopt "engines" packaged as libraries that provide those features.

G3D is similar to the 3D engines that you would find in a film or game company, but it has been tailored for research and education. In particular, G3D has a modular design that allows you to replace components with ones that you built yourself, and because the full source code is available it provides about 200k lines of sample code (in addition to the samples that are in the documentation).

See the latest version of the G3D manual [McGuire 2010] for detailed information about the library.

7 Working from Home

I only support working on the department Mac computers in TCL 126 and the Special Purpose Lab using Emacs, gdb, and g++/iCompile with G3D and the libraries it includes.

However, you are *permitted* to use any development tools (such as Xcode), computer (such as your own laptop), or operating system (such as Windows) in this course. Beware that if you run into trouble, I'm probably going to tell you to use the CS department computing environment.

G3D 9.00 beta for Windows / Visual Studio 2010 and OS X / gcc is available from the G3D Subversion server (which is different than the course subversion server). See http://g3d.sf.net for information. Make sure that you use the 9.00 version from source, not the public release 8.00 binaries. Installation and use instructions are included with the library

The Visual Studio 2010 Express IDE for Windows is a free download from Microsoft. The OS X developer tools including gcc are a free download from Apple.

The course subversion server is available outside the department and from off campus. Beware that deadline timestamps are based on the server's clock, not your client machine's clock.

G3D Windows and OS X are 100% compatible. For my own research I move the same code between Windows and OS X on a daily basis. So you should be able to move fluidly between IDEs and operating systems on the same project.

References

COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. 2008. Subversion complete reference. In *Version Control with Subversion*. O'Reilly, ch. 9. http://svnbook.red-bean.com/en/1.5/svn.ref.html. 1, 3

MCGUIRE, M., Ed. 2010. *The G3D 9.00 beta Manual*. September. http://graphics.cs.williams.edu/course/cs371/f10/G3D/manual. 1, 15

ROBERTS, A., 2009. Getting to grips with Latex - Mathematics, December. http://www.andy-roberts.net/misc/latex/latextutorial9.html and http://www.andy-roberts.net/misc/latex/latextutorial10.html. 1, 14

VAN HEESCH, D., 2010. Doxygen 1.7.1 manual. http://www.stack.nl/~dimitri/doxygen/manual.html. 1, 13

Index