# GiT Tut10

**Tasks set in tutorials should be finished in your own time!**

## Intro

Look at the presentation and/or watch the video on GiT and scheduling. Here we try to put an entire game together (so don’t worry if you don’t have time for all of it) and think about the schedule and tasks involved. Along the way we’ll look at text file loading and saving.

From now on you should use GiT as a simple backup mechanism to ensure you don’t lose any work. We won’t use its real power yet, the ability for multiple people to work on the same codebase together and track each person’s changes.

### Create a GiT repository and starter code

I use bitbucket, you can get a free repository that is private and can have up to five people in a team with your account limited to 2GB (I think). If you would rather use a different repository then that’s fine as long as it uses the GiT system.

1. Get [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t10\_hangman\_starter.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t10_hangman_starter.git)
2. Turn off visual studio and delete the .git folder (this is the local repository that links to mine online).
3. Create your own repository
   * Github – visual studio has support for github built right into the editor
   * Bitbucket – In a browser, create your own repository for it, call it something sensible and add a readme file and description explaining what is in it.
   * Clone that repository into an empty folder.
   * Add a .gitignore file – you don’t want to waste your free space and you only need .sln, .vcxprog, .cpp, .h, .gitignore to be in your repository. NOTE – you can add dlls and graphics files, etc., but you might run out of space.

<https://github.com/github/gitignore/blob/master/VisualStudio.gitignore>

* + Copy your tutorial into the folder and check it builds and runs OK (delete the now empty source folder you copied from).

If using github, try github desktop – quite a nice app

* + GiT menu -> Commit your work back to the local repository.
  + GiT menu -> Push your work to your remote repository

Note – watch the videos / read the presentations about GiT.

Cheat sheet: <https://rogerdudler.github.io/git-guide/>

You can download GiT command line at home (and it might work in the lab) for free and not rely on visual studio if you prefer.

https://gitforwindows.org/

|  |
| --- |
| NOTE - the next questions really wrap up the spiral development process and the game design process which both follow the same approach:   1. **PRE-ALPHA** phase. Hack around (fast experimenting) until something looks promising, basic core game loop. 2. Start looking for the optimal game design (this comes back to the GDD, who is the game for, how long do you want to play, what kind of experience did you have in mind). With any game design idea, there are a thousand different flavours of it. 99.9% of them are sub-optimal (they work, but a bit boring). Of the 0.1% of implementations, some are better for different audiences/platforms. 3. Every time you feel you've actually made some decent progress, stop, refactor (rewrite all the nasty hacky code). Be careful, it's tempting to just go through the hacky mess and try and tidy it up with better variable names, pulling bits out into separate functions. DANGER. Sit down with the TDD and a pencil, with what you now know, what would be the ideal way of writing this, the ideal data structure layout. Usually that requires starting again! 4. When you don't make decent progress with an idea, delete that hacky mess (get rid of the branch). 5. Keep iterating: test it until you get an idea, hack that idea in to see how it feels, usually those ideas are rubbish so delete them (that's why we are hacking), when it's a good idea, refactor the code (well engineered - it often means rewriting the new bit again properly), check it in to save it, update the GDD and TDD, repeat. 6. **ALPHA** phase. As the app starts to get juicy and you think it could be nearing completion, the testing gets more intense and you say "no more new features!" 7. Now we need to flesh things out, there's no more new features coming, just adding all the levels, getting the text right, feedback to player, audio, polishing our coal into a rough diamond. 8. **BETA** phase. You think it's finished. OK, it crashes strangely now and again. A few bits are mock ups from the artists, but you are sure it's almost done. Testing now involves people outside the team, the public. It's all about fixing bugs and tweaking or optimising the gameplay, getting it really juicy. Bugs are no longer about crashing or obvious problems, it's now about subtle things (e.g. "it's too hard in the early stages", "I get bored at the half way point", "I don't understand some of the instructions" or "There's no feedback if mistakenly do X"). |

|  |
| --- |
| **Update vs Render**  Any game needs to update as fast as it can 60fps minimum. The game maintains a virtual invisible logical simulation of the game – play object, enemies, bullets moving, etc. This simulation needs updating as fast as we can so things appear to react promptly and move around. However, we can do all that and nobody ever actually see it. This is often driven by user input.  Rendering also needs to update fast, here we take our simulation and use that information to render sprites and text on the screen to show the player some (or all) of that simulation as it updates.  If you have one function and inside that update the logic AND render things AND check user input then it becomes messy very quickly. It’s common to separate the code into Update(), Input() and Render() functions, sometimes this is referred to as the Model/View/Controller design pattern.  In games it gets a bit muddy because there can be two types of input:   1. Windows messages – sf::Event::TextEntered these are used when someone is typing something in, usually in an Input() function. 2. Fast instant input – sf::Keyboard::isKeyPressed(sf::Keyboard::Right) these are used when instant feedback is needed (steering, shooting, etc.), usually done with and in the logic update. |

|  |
| --- |
| **Finite state machine**  Most games have a similar structure:   1. Intro – display a logo, maybe run a video or non-interactive 3D animation, load in large slow resources 2. Play – run the game 3. Game over – explain how badly they’ve done   You tend to see matching functions, often expanding into structs (classes) as things get more complicated:   1. Intro() 2. PlayGame() 3. GameOver()   But how do you decide which function to run when? This is one use of a state machine, they can get complicated, but this is simple.  int mode = 0  //…  if( mode==0 )  Intro()  If(keyPress==ENTER) mode=1  else if (mode == 1)  PlayGame()  if(livesLeft==0) mode=2  else  GameOver();  if(keyPress==ENTER) mode=1;  else if(keyPress==ESCAPE) quit();  It’s just one variable and some if/else statements, but an enum class and a switch statement make it look elegant.  enum class Mode { INTRO, PLAY, GAME\_OVER };  Mode mode = Mode::INTRO;  //…  switch(mode)  case Mode::INTRO:  Intro(); break;  case Mode::PLAY:  PlayGame(); break;  case Mode::GAME\_OVER:  GameOver(); break;  default:  error(); |

### Hangman V1 - basic working game loop (pre-alpha)

Let's have a look at file loading and searching. The simplest game I can think of that uses those features is hangman. Let's prototype it, start simple, test it. Basic first hacked prototype spec:

1. Add UpdateWelcome() and UpdateRender(window) to display instructions and wait for the used to press space to play.
2. Pretend we've picked a random word from some library of words (just fake it)
3. Show the player the word they are trying to guess e.g. "\_ \_ \_ \_" use SFML, but just text only
4. Get a key press from them
5. If the letter is in the word, then show it "F \_ \_ \_ "
6. If it isn't in the word, then count the number of fails
7. Display the number of fails, if they get to 7 then it's game over
8. If they guess the word then show them "F R O G" and wait to quit
9. If they accidentally press the wrong key more than once, don’t count it against them. So remember each key they press by storing in a vector<char> and ignore any repeats.

Here’s a flowchart of one way you might check what they type in:



Once all that is working, you have hangman ☺ If you get stuck then look at [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t10\_hangman\_basicsol.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t10_hangman_basicsol.git) . If you didn’t follow the flowchart, check out this pseudocode.

**In pseudocode**

*global usedLetters array of 50 chars //hold every unique key pressed*

*//they’ve pressed a key so check it’s new and then check if they guessed correctly*

***Proc UpdateGuess***

*local usedBefore //bool*

*if key not zero then*

*set usedBefore = false*

*for each letter in the usedLetters array*

*if usedLetter = key*

*set usedBefore = true*

*end for*

*if usedBefore = false then*

*call CeckNewGuess()*

*endif*

*endif*

*global targetWord array of 10 chars //big enough to hold maximum size word to guess*

*global guess array of 10 chars //big enough to hold the correct letters as picked*

*//it’s a new letter so check if it matches any letters in the word they are trying to guess*

***Proc CheckNewGuess***

*local done, found //bool*

*local tc //char*

*local i //int*

*set done = true*

*set found = false*

*set i = zero*

*for each letter in targetWord*

*set tc = targetWord[i]*

*if tc = key then*

*set found = true*

*guess[i] = key*

*end if*

*if tc ≠ guess[i] then*

*set done = false*

*end if*

*end for*

*if done = true then*

*set won = true //game is over*

*end if*

*if found ≠ true then*

*set lives = lives – 1*

*if lives = zero then*

*set won = false*

*set gameOver = true*

*end if*

*end if*

*set usedLetters[numUsedLetters] = key*

*set numUsedLetters = numUsedLetters + 1*

Still confused, what about this milestone plan?

A screenshot of a computer

Description automatically generated

Trello is free and very useful for task lists and scrum boards, read left to right and top to bottom. Each list is one milestone. Breaking up a complex problem into sub tasks is classic good engineering. Jira is more sophisticated, Excel takes time to setup, but can do anything.

### Make an experimental branch

You can use a branch to work on some new feature without checking your work into the remote online repository i.e. so the rest of your team don’t end up with half done code. You can commit the branch though and keep it separate, if you want an off-site backup.

1. Make a branch
2. Do some work, make any real change to the code (next question?)
3. Commit your branch and check in the online repository that it is recorded
4. Merge your branch back into the main branch once the change is finished and you know it compiles and works OK.

### Use a hard coded word array

Create a string array with ~10 elements and use literal constant strings to initialise it, choose a word at random from your new ‘mini’ word library.

const int MAX\_WORDS = 5; //words in the library

const std::string WORDS[MAX\_WORDS]{ //library of words

"hello",

"goodbye",

"muppet",

"monkey",

"superman"

};

int tgtWordIdx = rand()%MAX\_WORDS;

cout << WORDS[tgtWordIdx];

If you get stuck [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t10\_hangman\_basicsol.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t10_hangman_basicsol.git).

|  |
| --- |
| **Stringstream**  As complexity grows we need to display more and more information, literal strings, variable strings and numbers. A stringstream is ideal for this.  stringstream sstr;  sstr << “Hello, my name is“ << name << “, I am “ << age << “ years old.”;  sstr << “\nI live in “ << town << “ in England”;  Text txt(sstr.str(), font, 20);  window.daw(txt);  This is more elegant that using lots of string objects. |

|  |
| --- |
| **Save and load**  You used streams before to print to the screen (cout) and read input (cin), a similar process works with files too.  Save some information:  void Game::RoundOver()  {  ofstream file("data/save.txt", ios\_base::out);  if (file.is\_open())  {  file << bestName << ' ' << bestScore;  assert(file.good());  file.close();  }  }  Load some information:  void Game::Init()  {  bestName = "";  bestScore = 0;  ifstream file("data/save.txt", ios\_base::in);  if (file.is\_open())  {  file >> bestName >> bestScore;  assert(!file.bad());  file.close();  }  } |

### Make it more game-like

Keep choosing random words, let them continue until they fail, save the name and highest score.

1. Ask for a name when the game starts, if you have a game save then load and display the previous best score and player.
2. Give them 7 fails per word, seven chances to make a mistake. Show this as “Lives left: 7” on screen. When it hits zero they’ve lost, if they guess the word, reset the lives left count.
3. Save the name and number of words guessed of the best player. Show this when they die:
   1. “Game Over, the best score is 5 words by Fezzy, try again.”
   2. “Game Over, you got the best score with 5 words!”

If you get stuck look at:

[**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t10\_hangman\_savesol.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t10_hangman_savesol.git) .

Finishing something off properly, polishing your game, adding graphics, play testing, updating design docs it takes time. Something like that is worth it for your portfolio though, employers like to see finished well engineered projects, even simple ones.

### Hangman V2 - make it look better (pre-alpha)

As you make mistakes and go through the 7 steps to a hanged man and game over, draw a basic stick figure. We can use shapes for the gallows and the figure, but add a little texture here and there to make it interesting. The hanged man should draw in, one part at a time, to show the seven steps to game over. Hack it fast, experiment. Then do it right, well-engineered.

A yellow emoji on a wooden structure

Description automatically generated

It probably took quite a while to get to this point – that’s OK, feel free to jump to the end and check out the last question.

### Optional - Hangman V3 - load some real words (pre-alpha)

In a new branch, try to load some text and extract the words from it. I've given you a copy of the bible as a text file, also the scrabble word list, but you can use anything. This might be a good point to properly engineer the previous code. For now forget the game, just focus on loading.

1. Load the text file into a simple (but huge) array, use unsigned char as there could be strange characters in the file and we don't want negative numbers. Use the debugger to check it looks OK.
2. Iterate through the array
   * If you find a non-alphabetic character, with an alphabetic character following it then this could be the start of a word i.e. a space proceeding the first letter, although it might not be a space
   * Start collecting those letters until you hit a non-alphabetic character
   * At this point you've got a word, discard if less than 4 characters and check it isn't too big (put some upper limit like 13 letters on it)
   * Save that word in a big array of words, maybe 10k maximum
   * Display them to verify everything is working properly – there’s probably some letter combination that doesn’t work, can you isolate the sequence that breaks your code, makes it easier to test fixes.

Now you have hangman and it can generate letters from real text, plus if you've used the bible then that's 4.5 million characters. You should be able to process it without a noticeable delay. Obviously you can use any large text.

### Optional - Hangman V4 - trying to make it fun (pre-alpha)

There's a difference between working and fun. This isn't fun, what design approaches can we take to make it fun? Notice how they are going to make the design get a lot more complicated! WE definitely want a well-engineered set of code at this point, save it, check it in. If we just kept on hacking, then this is the point where we’d start to lose control of the code.

OK, here's an idea. Let's assume hangman gets harder if the word length gets longer - is that even a correct assumption? So, take your big library of words, sort them into separate arrays, 4letter words, 5letter words, up to 13letter words. The bible should generate ~1000 in each group. The player is now told they have to guess ten words in increasing order of difficulty. If ever they get the 7-step hangman then it's game over. The hangman resets between each word. They get a score, they get +1 for a correct letter, -1 for an incorrect letter, +2 for the whole word. This should give the game more play time. Question - is the scoring appropriate, I don't want to make it harder, just more interesting, but it should be possible for the score to hit zero and then it's game over, immediately draw the hanged man. Hack around with different values until it starts feeling juicy (good).

1. Start simple and see if you can sort words into groups based on length, display or debug them to verify it’s working
2. Now alter the game to increase the length of the chosen word by one every time you are successful. Display “Level X” on screen.
3. Now it’s interesting, ask the player to type in their name when they eventually die (or reach max word length).
4. Store this name and score in a text file, load and display when the game starts, only store the best score.
5. Store the top 10 best scores and display them all when the game starts.

### Optional - Hangman V5 - still needs more juice (pre-alpha)

Let's try and tweak the scoring, so a correct guess gets +2 points, a failed guess gets -1 points. Don't let the player accidentally record the same key again e.g. you shouldn't be penalised if you press the same incorrect key twice. Show a list of the keys they've pressed that were wrong, this helps them avoid frustration. Hacking is appropriate until we home in on an ideal solution.

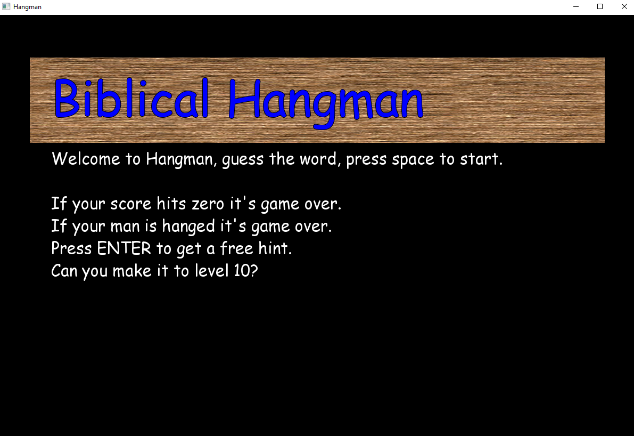
### Optional - Hangman V6 - avoid being screwed by chance (pre-alpha)

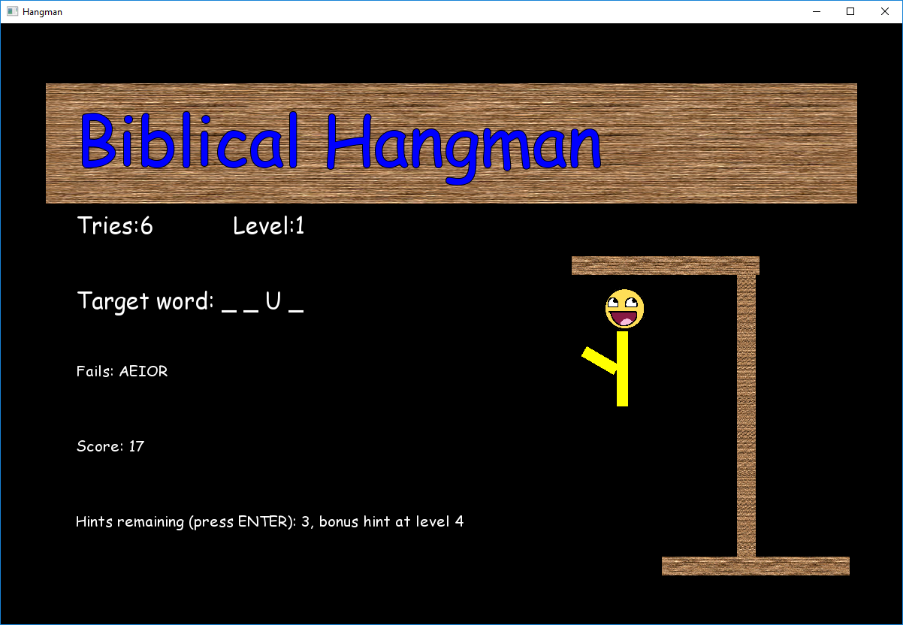
Now the game could actually last a while, but there's limited strategy (I just start with the vowels, always got to be one, then try 's' as it’s the most common start letter in English words) and what's worse, if you get a bad word then you can end up dead very quickly. Let's add a 'hint' option, you start the game with three hints, if you get stuck you spend one and a random hidden letter is revealed. You don't get any points for using a hint. You should get a reward of a new hint at certain levels, not all as that's too easy. Tell the player the next level to generate a new hint “bonus”.

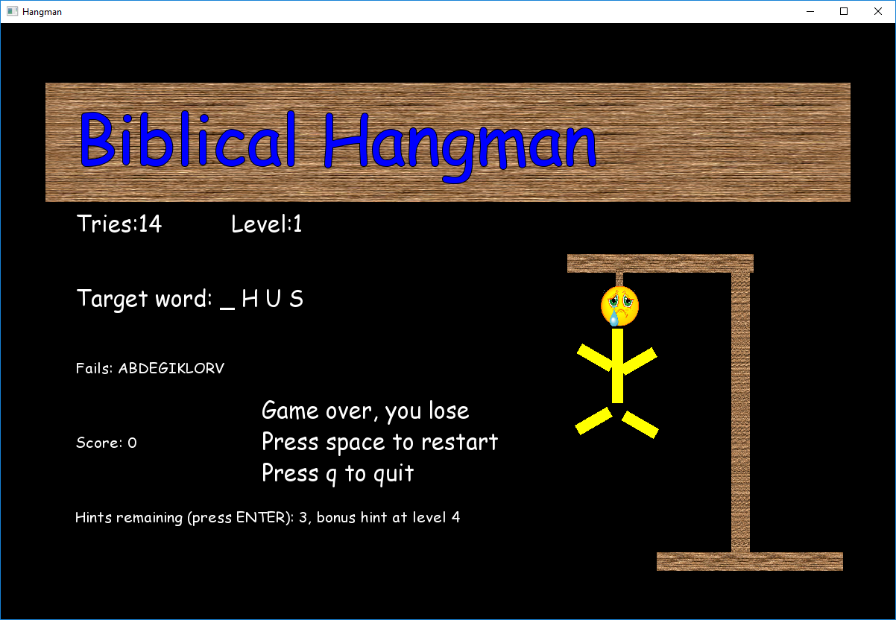
Now there's a new strategy, using your hints, plus you have something to play for as you know another hint is coming, it adds tension and something to look forward to. Still exploring and hacking around. It should feel more juicy now.

### Optional - Hangman - finished (alpha)?

What do you think? Is there anything else could be done to improve the game or is it now optimal in design? You must stop hacking if you haven't already and properly engineer the solution now. It should play well and in industry would now be ready for more intense testing or even a beta release - this must not be hacked code. It may be that we jump straight from pre-alpha to beta with this game as it's very simple and doesn't really have expanding content that we need to build. In most games we'd spend a lot of time making levels, items, new enemies, bosses, etc – this would be the alpha phase “feature complete”.







We could turn this into a franchise, a bible version, film versions, book versions, anything where the copyright is free or we pay the licence. Each one could have themed art.

Solution: [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t10\_hangman\_final.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t10_hangman_final.git)



****

### Create a schedule

One of the key reasons we make all those annoying design documents with all their system models is to get such a deep understanding of the problem that we can break it down into tasks on a schedule. Using the schedule you can spot:

* which tasks are risky (you aren’t sure how to do them)
* which are important and reorder them (key to the game’s U.S.P and need doing first)
* how to break the project’s tasks into sensible things for multiple people to work on (you can’t really split one thing between two people, you can’t all work on the audio)
* identify regular milestones (you are expensive and must convince your backers you are making good progress)
* how long the tasks are going to take which then adds up to how long the project will take (and how much it will cost)
* Spot the critical path, sometimes there may be 100 people on a project, but really there are three of them that are working on things absolutely critical to progress, that are holding up lots of other people (these tasks can be identified, split up, done earlier, people can be reassigned to non-dependant tasks until they are finished)
* Create a Trello/Jira/Excel schedule, think about the Hangman game, even if you didn’t finish, create a complete task list for it based on what you did and how long it took.
* Note time estimates for tasks. Your manager will want to know how long jobs will take, practice guessing. In a spreadsheet, write down the task names, your guess at how many hours, then write down how long it actually took later. Are you always one hour over? One hour under?

### Use GiT, branches and a schedule in your own work

1. Evidence with screenshots that you can use Git for personal backup. You can use your answers from earlier as we won’t have specific solutions anyway. Use screenshots to show the online repository commit history.
2. Make at least one branch and take a screen shot.
3. For scheduling, use Trello or Jira or a spreadsheet, soon we will turn our C code into C++. Use this tutorial or any project, but pick something a bit big or there’ll be nothing in the schedule. Record your task time guesses.

Competencies: 1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.9,2.1,2.2,2.3,2.6,2.7,2.8