Assignment 1 –Starter Code

Due Wednesday night (January 16th) by 11:59pm

# Tasks

* **Download** the starter codeand ensure it **runs**
* **Read** through the code to begin understanding the *overall architecture*
* **Read** this document and refer to it as necessary while looking through the code
* **Take** MyCourses survey by **posting** **questions** **you may have** about starter code

# Details

This document describes what you’ll need to have in order to run the starter code for the course. It also contains an overview of the starter code and each method you’ll find. Yes, it’s quite long. No, I don’t expect you’ll be an expert at it after reading through it once. But it’s here as a reference when something doesn’t make sense.

Download the provided starter code from MyCourses and ensure it compiles & runs on your primary work machine. This could be the IGM labs, a personal laptop, your desktop, etc. There is no code to write for this first assignment.

# Prerequisites

* **Visual Studio 2017**
* The proper version of the **Windows SDK**, depending on your operating system. This is where the latest DirectX 11 library files come from:
  + Windows 10 SDK: <https://dev.windows.com/en-us/downloads/windows-10-sdk>

# Deliverables

There is no specific *code* deliverable for this assignment. Instead, you’ll let me know via a **MyCourses survey** that you got it working, and include any questions you might have about the starter code itself.

The Starter Code

# Overview

The following pages describe the starter code and some of the architectural decisions I’ve made.

Where possible, it will also discuss other options you may have as well. There are plenty of places where I could have made different decisions, but I wanted to keep things simple. For example, if you prefer smart pointers to regular pointers, you could use them instead.

# Why do we need starter code?

If you’ve done any OpenGL programming, you’ve probably used a library like GLUT or GLFW to handle creating a window and initializing OpenGL itself. While there are a few similar libraries for DirectX, I decided a while ago to just do it myself and show you the code. I think it’s important to understand what goes into initializing our graphics API and creating the window (the guts of many of our programs).

Much of what you’ll see in the starter code is simply what libraries like GLUT or GLFW do for you. In fact, you could take the non-DirectX-specific parts of this starter code and use them to create an OpenGL application instead.

An advanced game engine might even initialize either OpenGL or DirectX based on a config file or command line parameters. However, this is *outside the scope of this course*, as it usually requires wrapping your entire graphics system in an abstraction layer or simply doing everything twice.

# Basic Architecture

The core architecture will consist of a few code files and a bit of inheritance:

* **Main.cpp** – Our program’s entry point
* **DXCore class** – Handles window creation, OS-level messages and DX initialization & shut-down
* **Game class** – Inherits from DXCore. Contains application-specific code.

The idea is that you shouldn’t have to touch DXCore at all (unless you’re fundamentally altering how it works). DXCore will take care of the basics that just about any DirectX game will need, and the Game class (plus any custom classes) will be where you actually build cool, custom stuff.

# Win32 Programming

A lot of the “windows programming stuff” you’ll see in the code might look a little foreign. It’s the Windows API (often referred to as Win32), and it hasn’t changed all that much in the last few decades, hence its archaic feel. Future assignments won’t interact with it much.

# The Project Itself

I started by creating an empty C++ project in Visual Studio 2017 (using the template called “Empty Project”). Since I already had the Windows 10 SDK installed, I didn’t have to alter anything else or include any other libraries. However, I did include a set of helper classes I wrote called SimpleShader, which I’ll get into a little later in this document and in the next few weeks of class.

# Main.cpp

This is where our program entry point resides. It turns on memory leak detection (in debug builds), ensures the program’s working directory (where its relative path begins) is the executable’s folder, creates our one & only Game class object, initializes it and enters our game loop. That’s about it.

However, since we’re creating a graphical Windows application, there’s a bunch of OS-level (Operating System – Windows in this case) stuff that our application must handle. It mostly gets dealt with by DXCore, but it’s worth talking about it a little here.

### Entry point: Main vs. WinMain

The first thing that may be a little different from what you’re used to is that we don’t have a main() function. Instead, we use WinMain(). This is a convention used by Win32 graphical applications that don’t require a console window (although we can allocate our own console later if we want).

One advantage of using WinMain() is that it comes with some useful OS-level parameters that we’re going to need anyway when setting up a window and DirectX. Of course, we could just use main() and get that information another way, but WinMain() is the standard way of starting this kind of app.

If you’re interested, you can find [more information about WinMain on MSDN](https://msdn.microsoft.com/en-us/library/windows/desktop/ms633559(v=vs.85).aspx).

### Application Handles

Every application (and window, and UI element, etc.) has a unique identifier assigned by the operating system. These are called *handles*, and we use them when we want to reference or create a particular object controlled by the OS, such as creating a window for our application to use. Under the hood, handles are just integers, although the data type we use is HINSTANCE.

Since we’re using WinMain(), it’s implied that we’re going to be creating a window for our program, so the first parameter is our application’s OS-level handle.

### S\_OK and HRESULTs

You’ll see the constant **S\_OK** in WinMain(). The “S” stands for success, and the “OK” stands for, well, okay. This is a predefined constant often seen with Win32 programming that means “Everything is ok, no errors occurred!” Really it’s just the value zero.

Most Win32 functions have a return value of type HRESULT, which is a 32-bit value divided into three sections (severity, facility and error codes). There are some predefined constants and macros in the windows API that will simplify checking these return values. S\_OK is one of these constants.

# DXCore Header

Before we get into the guts of DXCore, it’s worth looking at a few novel pieces of the header file.

### Including Library (.lib) Files

You’ll notice the following line of code near the top of DXCore.h:

#pragma comment(lib, "d3d11.lib")

What’s that **pragma comment** thing? Often we need to denote that we’ll be using a library that isn’t normally included in our project. This can be done through Visual Studio’s project settings, or by including this particular preprocessor directive directly in our code.

One advantage to putting it directly into the code is that it’s obvious: you can tell the library file is needed without having to search through settings. If you’re more comfortable including it in your project settings, you could do that instead.

### Windows Message Handling

You’ll also notice two static members in the public section of DXCore.h:

* DXCoreInstance
* WindowProc()

#### WindowProc

WindowProc() is a necessary part of creating a graphical windows application, regardless of whether we’re making a game or just a regular application.

Under the hood, every graphical Windows application is sent messages by the operating system. These messages tell the program things like “the mouse just moved”, “your window is being resized” or “the user clicked the close button”. Our program must deal with these, by either capturing them for our own use or by simply telling the operating system “ok, we got it, thanks”.

If an OS-level message isn’t dealt with properly, Windows thinks our program is frozen. This is how Windows detects that a program “isn’t responding” – it’s literally not responding to the messages being sent to it. This can happen when a program is stuck inside an infinite loop, for example.

So what makes WindowProc() special? It’s the function within our program that Windows will call when it wants to send us a message. However, it **cannot** be a member function! This means it has to be a global or static function, which makes it harder to talk to our non-global Game object.

#### DXCoreInstance

The quick and dirty answer to the problem above is to create a static pointer to our game object, called DXCoreInstance, and set it during the object’s constructor. This will let WindowProc() talk to our object.

If this makes you cringe a little, that’s probably good. Like I said, it’s a quick fix without adding a bunch of extra complexity to the starter code. It might be a little safer to use a singleton or similar design pattern, but for our purposes this will work fine.

# DXCore Class

This class is the core of our little engine. It’s going to create a window for our application, initialize DirectX, handle incoming Windows messages and perform a few other odds & ends. Descriptions of major areas and methods can be found below.

### Includes

Aside from the obvious DXCore.h, we’re including the following two headers:

* **WindowsX.h**: Needed for a few of the windows messages we plan on handling
* **sstream**: Used for quick and dirty debug string manipulation in UpdateTitleBarStats()

### Static Members

Here we *define* DXCoreInstance, since we’ve only *declared* it over in the header file.

Directly below that, you’ll find the WindowProc() function. It forwards its parameters to our object’s ProcessMessage() function, which will actually check the message and do something useful if necessary.

### DXCore Constructor

The constructor for DXCore takes a few parameters related to the window we’ll be creating:

* **hInstance**: The applications handle, which is needed to create a window
* **titleBarText**: The text you want in the title bar
* **windowWidth**: The width of the window’s *client* area (not including title bar & borders)
* **windowHeight**: The height of the window’s *client* area
* **debugTitleBarStats**: Should extra stats be shown in the title bar, such as FPS and DX version?

The constructor will set up the global DXCoreInstance pointer, save parameters and initialize a few fields. (Initializer lists are another option here.)

Lastly, it queries the *performance counter* to find out its frequency. The *performance counter* is a way of retrieving very high resolution time stamps (less than 1 microsecond), allowing us to calculate very precise timing. The frequency may be different on each machine, but is fixed at boot time, so we only need to grab it once at start up.

### DXCore Destructor

Releases any DirectX resources we’ve created. Rather than deleting them directly, we call Release() on them. Each has an internal reference count, and once the last reference is released the object is automatically cleaned up.

### DXCore::InitWindow

This method creates the window for our application, which must exist before we can initialize DirectX.

The first thing you’ll notice is a big struct of type WNDCLASS being filled out. This will hold the basic settings for the window class we’re trying to register with the operating system, and is passed to RegisterClass(). We can’t create a window unless we register one of these special window classes that describes it. Some options at this step include which function to call when messages come in, the icon for the application and default cursor to show.

If that succeeds, we then create a rectangle of the appropriate window size and adjust it to include borders and the title bar. We also calculate the centered X and Y location to be used in the next step.

Next up, we actually call CreateWindow(), passing in the name of the window class that was registered, some information about the size of the window and its title bar text.

As long as the call to CreateWindow() succeeded, we show the window (finally making it visible) and return S\_OK to indicate success.

### DXCore::InitDirectX

This method initializes DirectX, which requires a window to already exist. In addition to simply “activating” DirectX, this method creates the basic resources we’ll need to do anything useful with DirectX, including creating the *back buffer* (where we draw) and a *depth buffer* (for basic occlusion).

First though, a preprocessor directive determines if we’re in Debug mode (rather than Release mode), and sets the D3D11\_CREATE\_DEVICE\_DEBUG flag. This causes DirectX to do a lot of extra error checking and provide (somewhat) useful error messages at run time. Since this can slow things down a little, we only want this to happen while in debug mode.

It then creates the description of the *Swap Chain*, which is an object that automatically handles double buffering (sometimes called *page flipping*) for us. Options here include the size of the buffer (the image to be presented to the user), as well as the color format and whether or not the application should be windowed.

Now that we have that description, we can call D3D11CreateDeviceAndSwapChain(), which initializes the big three objects we use to communicate with the Direct3D API:

* Device – Creates resources on the GPU
* DeviceContext – Issues drawing commands and state changes (called *context* for short)
* SwapChain – Handles presenting our final frames to the user

One useful feature of DirectX 11 is that it can automatically fall back to older versions if it detects incompatible hardware. One of the parameters to D3D11CreateDeviceAndSwapChain() will hold the actual feature level being used once the function returns.

The creation of the swap chain automatically creates the back buffer resource, but we need a specific wrapper (called a *view*) around it so we can use it correctly. This means we need to get a reference to the buffer from the swap chain and build a *Render Target View* for it. This is essentially a wrapper that denotes we plan on using this particular image as a *render target* (putting new pixels in), rather than as a *texture* (pulling pixels out).

Next up, we create a *depth buffer*, which handles basic depth-based occlusion for us (*not* sorting! A depth buffer doesn’t sort anything). Again, we fill out a large struct with the options we want, and then pass it to a method that creates the resource. This is a very common patterns with DirectX and Win32 programming. A few of the bits of each pixel in a depth buffer are sometimes used for a feature called stenciling, so the full name of the view is the Depth Stencil View.

You may also notice that we seem to be creating (or getting existing) textures, creating views for them, and then releasing the textures. This doesn’t immediately destroy the texture, it simply releases one reference that the texture knows exists. Each view we make also adds a reference to the texture. Since we generally only interact with the views, we can release our texture references immediately, preventing leaks later on.

Lastly, we set the current render target and depth buffer, and create a viewport that defines how much of the window we actually want to render into. Our viewport will be the entire window, although you could create multiple and swap between them while drawing to create split-screen games.

### DXCore::OnResize

The resolution of our buffers (like the back buffer and the depth buffer) are tied to the size of the window. So what happens when we resize the window? If we do nothing, the window’s resolution and our rendering resolution won’t match.

OnResize() is called by another part of DXCore when we get a message from Windows telling us the window size has changed. It releases any existing views into GPU resources, resizes and/or recreates those resources, and wraps them in new views. It also updates the viewport to match, so we always fill the whole window.

### DXCore::Run

Run() is the last thing called from our WinMain() function, after the window and DirectX are ready. It’s where our actual game takes place, and where OS-level messages are first processed.

The first thing that happens in Run() is we get a timestamp that represents the “start” of the game, so we can measure total time during the game loop. Next, the pure virtual Init() method is called, allowing a subclass the chance to do any initialization logic.

Then we enter the overall game and message loop, which is a while loop that continues until it finds a WM\_QUIT message. This message could come from several sources: closing the window, pressing Alt+F4 or by adding our own quit message to the message queue.

Inside this loop, we first determine if a message is waiting to be processed by peeking at the queue. If one is there, we remove it from the queue and handle it. It is first *translated* (this alters some keyboard input to be more appropriate), and then *dispatched* to our static WindowProc() function.

How does it know to call WindowProc() during dispatch? Remember: A function pointer to WindowProc() was required as part of the creation of the window, back in InitWindow().

If there are no messages waiting to be handled, our game loop proceeds: We update the timer, and potentially the window’s title bar, and then call Update() & Draw().

Once we get a quit message, the loop ends and we return the result from that message. Since Run() ends, so does our WinMain() function and our overall program.

### DXCore::Quit

Sometimes we want to be able to quit through code, like when the user clicks “exit” on a menu or we detect an unrecoverable error. Since our game loop ends when we get a quit message, we can simply post our own quit message and our program will end gracefully.

### DXCore::UpdateTimer

Each time through our game loop, we want to know how much time has passed, both since last frame and since the beginning of the game. UpdateTimer() queries the performance counter to get the latest timestamp, and then calculates those pieces of information.

Both deltaTime and totalTime are stored as floats, since they’re most often used with our math library’s vectors and/or matrices, all of which use floats instead of doubles. This saves us from having to cast from double to float over and over later on. Since they’re fully recalculated each frame, we shouldn’t run into any round-off errors.

### DXCore::UpdateTitleBarStats

This method counts frames until 1 second has gone by, and then updates the title bar with:

* Width and height of the current window
* Current FPS
* Approximately how long each frame took (ms/frame)
* The version of DirectX that is currently in use (usually 11)

This is not the most performant function, but it’s very useful to gauge *relative* performance. If you add a feature and see your framerate drop drastically, you know where to start optimizing.

One thing to remember: **Don’t judge your overall performance while in debug mode!** Try running your game in release mode and check the FPS. Chances are it will be vastly higher, as DirectX won’t be in debug mode and functions marked as inline will actually be inlined.

Seriously, if you’ve only ever looked at the performance of libraries like STL in debug mode, you haven’t seen the complete picture.

### DXCore::CreateConsoleWindow

Since our program is a Win32 graphical application, it doesn’t have a console attached to it. However, it’s often useful to have a console available while debugging. To that end, this method creates a console and redirects the stdin, stdout and stderr streams to it.

It also disables the close button on the console window, as closing the console also closes the overall application. If that’s desirable behavior to you, comment out the last 3 lines in this method.

### DXCore::ProcessMessage

This is the member function that is called from our static WindowProc() function, allowing us to both handle messages and interact with our game loop. The majority of this function is a big switch statement, each case handling a different message.

When we handle a message ourselves, we return zero to inform the OS that we’re done with it. If, however, it’s a message we don’t care about, we still have to report that to Windows. Hence the final line of this function, DefWindowProc(), which tells Windows to use the default processing for this particular message.

Below you’ll find a quick discussion on the messages this method handles:

* **WM\_DESTROY**: The window is closing, so we post a quit message
* **WM\_MENUCHAR**: Alt was pressed, suppress beeping for incorrect key combinations
* **WM\_GETMINMAXINFO**: Prevent the window from getting too small
* **WM\_SIZE**: The window size has changed, so we need to resize
  + We get a constant stream of these messages while dragging a window’s border
  + Performance also tanks since we’re constantly resizing buffers on the GPU
  + It’s a huge pain to try to get the window to update “live” during a resize
  + You won’t see anything new being drawn until you release the mouse
* **Mouse related messages**: Free mouse input, since windows tells us this info anyway
  + These messages cause mouse-related virtual methods to be called
  + These can be overridden in child classes to handle mouse input

# Game Header

Game.h includes the following headers:

* DXCore.h – Since Game inherits from DXCore
* SimpleShader.h – Helper classes I wrote for simplifying shader variable manipulation
* DirectXMath.h – The math library we’ll be using

I’ll be discussing individual methods in the next section.

# Game Class

This class houses our application-specific code. DXCore has all of the stuff *every game* needs, so Game will have the stuff *our game* needs. This is where you can really begin customizing.

Much of the code in here will be expanded upon, altered and sometimes even replaced in future assignments. Its current state represents the bare minimum needed to get a single triangle on the screen. It may seem like a lot, and it is, which is why we have starter code.

Is inheritance necessary here? Nope! But it creates a logical delineation between the “core” DirectX bits and the “game” code. Once the group project starts, refactor to your hearts content.

### Includes & Namespaces

Obviously we need Game.h. We’re also including Vertex.h, which defines the structure of vertices we’ll be sending to the GPU. Since this will be used in several places in the future, it makes sense to separate it out into its own file.

We’re also using the DirectX namespace, as that is where our math library resides.

### Game Constructor

The constructor receives the application’s unique handle when created in WinMain(), and passes it to the base constructor. Also being passed to the base constructor are the title bar text, window size and a Boolean for showing additional stats. Feel free to customize!

Internally, the constructor initializes some pointers to null and creates a console window (only in debug mode). If you don’t want the console, or you’d like it to exist in release mode for some reason, here is where you can change those things. You can edit or remove the sample printf() call as well.

### Game Destructor

Releases DirectX resources specifically created by the Game class, and the SimpleShader wrapper objects. **You are responsible for cleaning up your memory leaks and unreleased DirectX resources!**  You will lose points if you turn in assignments with memory leaks (or warnings, for that matter).

### Game::Init

Called exactly once by DXCore after the window and DirectX are ready, but before the game loop begins.

It currently calls three helper methods, each of which does a piece of the basic initialization for the game. You should alter, add to or otherwise adjust these to suit your own needs in later assignments or the group project. The three methods are described in the following sections.

The last thing Init() does is set the default primitive topology for rendering. This says “Hey DirectX, I want you to render triangles”. Since DirectX saves state, you only need to set this once, unless you plan on rendering multiple different types of primitives (like triangles and lines). However, completely removing this line of code will have different results based on your GPU manufacturer and current drivers (defaulting to triangles on some machines, points on others, and sometimes hard crashing the GPU drivers on yet others).

### Game::LoadShaders

DirectX is a fully-programmable rendering pipeline. This means to draw anything with DirectX, you must use shaders. There’s literally no other option.

Our shaders (stored in .hlsl files) are actually compiled for us into compiled shader object files (.cso files) at build time in Visual Studio. Visual Studio will even detect and highlight syntax errors, provide line numbers, etc. It’s quite nice. So we write .hlsl files, but load .cso files. Easy, right?

DirectX comes with everything we need to load and use shaders. However, it is designed to deal with large groups of shader variables at once rather than individual variables (as it is far more efficient to copy lots of data to the GPU once, rather than little bits over and over). This means there’s no way to just say “Hey shader, set this one particular variable to this specific value”.

This is great for efficiency, but it sucks for learning and experimenting, as it requires us to write a lot of extra C++ code to make even small changes. To this end, I’ve created a set of helper classes called SimpleShader, which wrap and simplify shader variable manipulation.

I’ll be getting into the details later in the semester, but the short version is that SimpleShader uses shader reflection to determine what variables exist, then builds tables (unordered maps) of variable names and their positions in memory. It then allows you to set individual variables by storing the data locally, and can send it to GPU as one big chunk later. The take away is that it’s a huge time saver as far as writing code, and ends up being more efficient than sending data to the GPU one variable at a time.

Why am I mentioning all of this? Because in LoadShaders(), we create two SimpleShader objects and use them to load the shaders. We can interact with the shaders later on by going through these objects.

### Game::CreateMatrices

When drawing objects in 3D, you need a matrix to represent all of the transformations of the object. In DirectX lingo, this is the *world matrix* (called the “model matrix” in OpenGL). We also need matrices that represent the camera’s location/orientation (the *view matrix*), and its “lens” (the *projection matrix*).

The method creates these three matrices using DirectXMath, and stores them in a way that is useful to our shaders. It’s ok if this code doesn’t appear to be straight forward (yet).

DirectXMath is a SIMD optimized math library, which means it can take advantage of newer CPU features to speed up our calculations. However, this comes at the price of complexity: there are some rules we have to follow in regards to how we use and store the data, or our code may end up slower than before (or completely broken).

You may notice that we’re transposing the matrices before storing them (and yes, I know it’s redundant on the identity matrix). This is done because DirectXMath uses row-major matrices (due to the SIMD optimizations), but HLSL uses column-major matrices. It’s an extra step, but one that you’ll need to do.

### Game::CreateBasicGeometry

If we want to draw something on the screen, in this case a triangle, we need the data to do it. Here we’re defining that data: 3 vertices and 3 indices.

Each vertex is made up of a position in 3D space, and an RGBA color. This exact definition can be found in Vertex.h. We then create a DirectX GPU resource called a Vertex Buffer to store the vertices directly in GPU memory.

One option when drawing is to also provide indices into our vertex list, so that neighboring triangles can potentially share vertex data (reducing the overall size of our memory footprint). While it’s not strictly necessary for such a simple example, we’re creating indices and an Index Buffer to showcase how they work.

The vertex and index buffers will be used later to actually draw the geometry.

### Game::OnResize

Our game class needs to be notified of window resizes so it can update the aspect ratio of the project matrix as necessary. We also need to call the base class version to ensure everything else gets updated.

### Game::Update

This is the first half of the game loop, and is where all of your game logic and input should be handled. For now, it simply checks for the escape key and quits if necessary.

### Game::Draw

This is where we’ll actually put stuff on the screen. There are a few things we need to do every frame, regardless of what we’re drawing. Much of the code here will be moved elsewhere in future assignments as you create game entities and material systems.

First we define a color to clear the screen. Cornflower Blue is given, although you may change it. We then call DirectX methods to clear both the Render Target View (the colors of the pixels) and the Depth Stencil View (the depth of each pixel).

Next up, we need to get the shaders ready for drawing. We’ll send our current matrices to the vertex shader by first setting the individual variables locally, and then calling CopyAllBufferData() to do a single copy to the GPU.

We also activate these shaders by calling SetShader() on each. We don’t necessarily have to do this every frame at the moment, since DirectX saves state, but when we do need to swap shaders it will happen in Draw(), so it makes sense to demo it here.

Now it’s time to activate a particular set of vertex and index buffers. Again, since we only have one of each at the moment, we could just set them once. But it’s a demo.

We can now call DrawIndexed() to tell the GPU to use the currently set vertex buffer, index buffer, shaders and other options to kick-start the rendering pipeline and get some pixels on the screen. We’ll need to call DrawIndexed, or some other draw command, each time we want to draw something to the screen. We do not have to re-set resources that remain the same between our draws, such as shaders (in fact, doing so is inefficient).

Lastly, the user won’t see anything until we tell the swap chain to Present(). This should be done only once per frame, after we’ve drawn everything we want the user to see.

### Game’s Mouse Input Methods

These methods are provided as a simple source of mouse input, since our window is receiving mouse input messages anyway. The names should be self-explanatory, but there are a few notable items.

You’ll notice that some functions call SetCapture() and ReleaseCapture(). What do these do? When our application captures the mouse, it continues to receive mouse input messages even when the mouse *goes outside the bounds of the window*! This will let you click and drag all around your screen and still get mouse input.

What exactly is “buttonState”? It’s a bitwise flag with several possible values indicating which buttons are being pressed ([this MSDN article](https://msdn.microsoft.com/en-us/library/ms645607(v=vs.85).aspx) has the list of possible button state values, under *wParam*).

For example, to check if the left button is pressed, use a bitwise and:

if( buttonState & MK\_LBUTTON ) // Left button is down

# Other Files

There are several other files in the starter code:

* **Vertex.h**: Contains the definition of our basic vertex data
* **SimpleShader.h**: Header for my custom shader wrapper
* **SimpleShader.cpp**: Code for my custom shader wrapper
* **PixelShader.hlsl**: Basic pixel shader for drawing (OpenGL calls these Fragment Shaders)
* **VertexShader.hlsl**: Basic vertex shader for drawing

These will each be discussed in more detail in class and in future assignments.

# Pheww, now what?

If you made it here, that’s awesome. You probably still have some questions though. Please ask them in class (or via email if they’re imperative), as other students might also benefit from the answers.