Dom Pedro II's Mod Helper

Getting Started

This is a simple tutorial for using the Mod Helper mod component. This mod does not modify the game by itself. It's intended as a way to easily allow modders to make more extensive changes to their own mods using python. My changes are based entirely on the preexisting code for the EventInfos. For an working example of how my improvements work, see the CIV4EventInfos.xml and the CvRandomEventInterface.py files.

Why is this tool useful?

There are a few things that this mod component allows modders to change that weren't possible before, but it is really more about giving people better ways to do things that would previously require a lot more python coding that would require more work from modders and slower processing time because python is slower than C++.

By allowing modders to tie particular python methods to techs, specialists, units, promotions, and other info types, it removes the need to have modders designing code to cycle through players, units, and cities, and allows them to create simple self-contained methods that will be accessed from inside the DLL only when necessary.

That's great, but what do I do with this?

There are three main components to this mod: XML, Python, and the CvGameCoreDLL. They work in concert with each other here just like they do for the regular game.

Python: Python is the main workhorse of this mod. It's with Python that you will be getting the game to actually do something. Python works in blocks of code called "methods".

The XML: You will not need to use all of the XML files provided in this modcomp if you're only making to changes to one or two of them. The XML is necessary to tell the game where to look for the python methods you want to execute. The XML is what connects the Python code to the DLL.

CvGameCoreDLL: When the game processes things like buildings, promotions, specialists, etc. in the DLL, it will call any python methods defined in the XML.

Relevant Files and Tags:

CvPythonDLLInterface.py

This file contains all of the python methods that are defined in the XML files below.

CIVSpecialistInfos.xml

<iAIWeight/>

<PythonCallback/>

<PythonAIWeight/>

<PythonHelp/>

CIV4TechInfos.xml

<PythonCallback/>

<PythonFirstText/>

<PythonHelp/>

CIV4PromotionInfos.xml

<iAIWeight/>

<PythonCallback/>

<PythonDoTurn/>

<PythonAIWeight/>

<PythonHelp/>

CIV4UnitInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonAIWeight/>

<PythonAIUpdate/>

<PythonHelp/>

CIV4CivicInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonHelp/>

CIV4BuildingInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonAIWeight/>

<PythonHelp/>

CIV4CivilizationInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonHelp/>

CIV4LeaderHeadInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonHelp/>

CIV4TraitInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonHelp/>

CIV4BuildInfos.xml

<PythonCanDo/>

<PythonCallback/>

<PythonHelp/>

CIV4GoodyInfos.xml

<PythonCanDo/>

<PythonCallback/>

CIV4BonusInfos.xml

<PythonCallback/>

CIV4VoteInfos.xml

<PythonCallback/>

CIV4ProjectInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonAIWeight/>

<PythonHelp/>

CIV4ReligionInfos.xml

<iAIWeight/>

<PythonCityCallback/>

<PythonCityDoTurn/>

<PythonStateReligionCallback/>

<PythonStateReligionDoTurn/>

<PythonAIWeight/>

<PythonHelp/>

CIV4CorporationInfos.xml

<PythonCallback/>

<PythonDoTurn/>

<PythonHelp/>

As you might have noticed, most of the files have four new tags described below:

PythonCallback - A python callback points to a python method that is executed whenever the particular item is processed in the game. This could be when a civic is adopted, a promotion is acquired, a goody hut is explored, a building is built, etc.

PythonDoTurn - Methods defined in this tag will be executed every turn rather than when they are processed. This can be used to have an effect occur every turn or every few turns using conditions.

PythonAIWeight – Methods defined in this tag will be executed when the AI is assessing the value of the particular item. There is already a static AIWeight value in the XML for most game items, so it is recommended that you use this for anything that has a consistent value as opposed to something that might only be valuable under certain conditions.

PythonHelp - Methods defined in this tag can be used to create help text for tooltips and the civilopedia that's indistinguishable from regular help text.

Other method tags:

PythonAIUpdate – Only found in CIV4UnitInfos.xml. Methods defined in this tag will be executed every turn for specified units. This allows modders to control AI behavior for particular unit types.

PythonCityCallback – Only found in CIV4ReligionInfos.xml. Methods defined in this tag will be executed whenever a religion is added or removed from a city.

PythonCityDoTurn – Only found in CIV4ReligionInfos.xml. Methods defined in this tag will be executed every turn that the specified religion is in the city.

PythonStateReligionCallback – Only found in CIV4ReligionInfos.xml. Methods defined in this tag will be execute when the religion is adopted as the state religion. It will continue to apply even when running a civic that does not have the State Religion tag (such as Free Religion).

PythonStateReligionDoTurn – Only found in CIV4ReligionInfos.xml. Methods defined in this tag will be executed every turn for as long as this religion is the state religion.

PythonFirstText – Only found in CIV4TechInfos.xml. Method defined in this tag is used to compose a message to be displayed to all players that have met the first discovering player of a particular technology.

New Content

Action Buttons

Many mods make include unit actions specific to that mod. Most of these new actions are added with python, and all of the mods and scenarios included with Civilization 4 and its expansions make use of python to add new action buttons. Adding buttons in python from scratch can often be tricky and requires at least a moderate understanding of python.

This mod includes along with it a heavily-modified version of talchas's Action Buttons 2.0 that allows modders with little python experience to add in their own action buttons.

Mod Helper includes a new XML file called **CIV4ActionButtonInfos.xml** where all new Action Buttons are defined.

```
<ActionButtonInfo>
        <Type>ACTIONBUTTON_TEST_ACTION</Type>
        <Description>TXT_KEY_ACTIONBUTTON_TEST_ACTION</Description>
        <Civilopedia>TXT_KEY_PEDIA_ACTIONBUTTON_TEST_ACTION</Civilopedia>
        <Strategy>TXT_KEY_STRATEGY_ACTIONBUTTON_TEST_ACTION</Strategy>
        <Help>TXT_KEY_ACTIONBUTTON_TEST_ACTION_HELP</Help>
        <EntityEventMission>NONE</Entit
        <EffectType>NONE</EffectType>
        <bPickPlot>0</bPickPlot>
        <iRange>0</iRange>
        <PythonCanDo/>
        <PythonCallback/>
        <PythonCanDoAtPlot/>
        <PythonCallbackAtPlot/>
        <PythonHelp/>
        <HotKey></HotKey>
        <bAltDown>0</bAltDown>
        <bShiftDown>0</bShiftDown>
        <bCtrlDown>0</bCtrlDown>
        <iHotKeyPriority>0</iHotKeyPriority>
        <Sound/>
        <Button>,Art/Interface/Buttons/Builds/Bui
</ActionButtonInfo>
```

EntityEventMission controls the unit animation when the action is performed. Any pre-existing Mission Types can be used here.

EffectType applies the effect to the affected tile

bPickPlot specifies that this action must have a selected target tile

iRange specifies the range of an action

PythonCanDo is used to determine if this action can be done by this unit on this tile

PythonCallback executes the action on this tile

PythonCanDoAtPlot deteremies if a target tile is valid for this action

PythonCallbackAtPlot executes the action on the target tile

Event Triggers

The Events system included in BtS offers a versatile and powerful system for creating game events, but it is not without its problems. In order to create a number of events, the modder would have to create many individual event triggers rather than simply use one and have python methods select the particular conditions of the event.

CIV4EventTriggerInfos.xml:

```
                <PythonProbability/>
```

```
<PythonCityTriggerValue/>
<PythonPickTech/>
<PythonPickBuilding/>
<PythonPickCivic/>
<PythonPickBonus/>
<PythonPickReligion/>
<PythonPickCorporation/>
<PythonPickCulture/>
<PythonPickOtherPlayer/>
<PythonText/>
```

TriggerImages – TriggerImages works exactly like the TriggerTexts. The game will randomly select one of the defined images for the proper era. If an era is not defined with the image, it will always be eligible to be selected. The image will then be displayed above the body text in the event popup.

TriggerSounds - TriggerSounds works exactly like the TriggerTexts. The game will randomly select one of the defined sounds for the proper era. If an era is not defined with the sound, it will always be eligible to be selected. The sound will be played when the event popup appears.

PythonProbability – The method defined here is executed when the game determines the probability of a particular event trigger.

PythonCityTriggerValue – The method defined here can modify the value of a city to be triggered. By default, trigger cities are determined randomly from among valid cities.

PythonPickBuilding – The method defined here will pick a building in the trigger city rather than having it chosen at random.

PythonPickTech – The method defined here will pick a Technology.

PythonPickCivic – The method defined here will select a civic that the player can adopt if they select

PythonPickBonus – The method defined here will select a particular resource.

PythonPickReligion – The method defined here will pick a religion rather than randomly selecting one.

PythonPickCorporation – The method defined here will pick a corporation.

PythonPickCulture – The method defined here will pick a culture in the trigger city. This could be useful if you want to make an event that concerns a foreign population in a captured city.

PythonPickOtherPlayer – The method defined here will pick another player involved in the event rather than have the player selected randomly in case the modder would like an event that is limited to players that share a border, ones that have a bad attitude with the human player, or any other criteria the modder wishes.

PythonText – The method defined here will return text to appear in the event popup. Under most circumstances, this will not be needed, but text created in popup gives the modder greater flexibility.

Events

The power of Events is already expanded by the new tags in the EventTriggerInfos, but EventInfos has some new tags of its own.

CIV4EventInfos.xml:

```
<PythonBestTech/>
<PythonBestCivic/>
<PythonGold/>
<PythonText/>
<PythonAIValue/>
```

PythonBestTech – The method defined in this tag returns a tech to be awarded to the player upon choosing this event.
PythonBestCivic – The method defined in this tag returns return a civic to be adopted by the player upon choosing this event.
PythonGold – The method defined in this tag returns the gold cost (or reward) for the event. This is allows modders to have the gold fluctuate depending on circumstances.
PythonText – Currently there is only one set of text for events, but this will allow modders to create their own text in python.
PythonAIValue – Adds or subtracts additional value for the AI depending on current circumstances

Tutorials

I'm going to demonstrate the power of this mod component with several simple modifications. In this tutorial we will do the following:

- Have Merchants produce +1 Happiness with Furs and Spices.
- Have a Wonder that produces 1 Maceman every 10 Turns.
- Modify Pacifism to give +2 culture per State Religion Building
- Create a Build option for Workers that will build Catapults out of Forests
- Create a new unit Action to heal a Unit
- Create a new unit Action to spread unhealthiness in an enemy city using Catapults

Warning: If you decide to copy and paste directly into the relevant game files, make sure that the code in the python functions are indented with tabs and not spaces or you WILL get errors. Consider yourself warned.

Tutorial 1: The Merchant

The first step is to define our python methods in the XML. We have to do this in two places

CIV4SpecialistInfos.xml:

```
<SpecialistInfo>
    <Type>SPECIALIST_MERCHANT</Type>
    <Description>TXT_KEY_SPECIALIST_MERCHANT</Description>
    <Civilopedia>TXT_KEY_CONCEPT_SPECIALISTS_PEDIA</Civilopedia>
    <Strategy>TXT_KEY_SPECIALIST_MERCHANT_STRATEGY</Strategy>
    <Help>TXT_KEY_SPECIALIST_MERCHANT_HELP</Help>
    <Texture>Art/Interface/MainScreen/CityScreen/merchant.dds</Texture>
    <bVisible>1</bVisible>
    <GreatPeopleUnitClass>UNITCLASS_MERCHANT</GreatPeopleUnitClass>
    <iGreatPeopleRateChange>3</iGreatPeopleRateChange>
    <Yields/>
    <Commerces>
        <iCommerce>3</iCommerce>
        <iCommerce>0</iCommerce>
```

```
                        <iCommerce>0</iCommerce>
                    </Commerces>
                    <iExperience>0</iExperience>
                    <iAIWeight>0</iAIWeight>
                    <Flavors/>
                    <HotKey/>
                    <bAltDown>0</bAltDown>
                    <bShiftDown>0</bShiftDown>
                    <bCtrlDown>0</bCtrlDown>
                    <iHotKeyPriority>0</iHotKeyPriority>
                    <PythonCallback>doMerchantCallback</PythonCallback>
                    <PythonAIWeight/>
                    <PythonHelp/>
                    <Button>Art/Interface/MainScreen/CityScreen/merchant.dds</Button>
                </SpecialistInfo>
```

In the Furs and Spices entries in **CIV4BonusInfos.xml:**

```
    <PythonCallback>doFursCallback</PythonCallback>

    <PythonCallback>doSpicesCallback</PythonCallback>
```

The next step is to create the python methods themselves in the **CvPythonDLLInterface.py** file.

```
def doMerchantCallback(argsList):
    pCity = argsList[0]
    iSpecialist = argsList[1]
    iChange = argsList[2]

    iFur = CvUtil.findInfoTypeNum(gc.getBonusInfo, gc.getNumBonusInfos(), 'BONUS_FUR')
    iSpices = CvUtil.findInfoTypeNum(gc.getBonusInfo, gc.getNumBonusInfos(), 'BONUS_SPICES')
    iHappiness = 0

    if (pCity.hasBonus(iFur)):
        iHappiness = iHappiness + 1
    if (pCity.hasBonus(iSpices)):
        iHappiness = iHappiness + 1

    pCity.changeBonusGoodHappiness((iHappiness * iChange))

    return

def doFursCallback(argsList):
    pCity = argsList[0]
    iBonus = argsList[1]
    iChange = argsList[2]

    iMerchant = CvUtil.findInfoTypeNum(gc.getSpecialistInfo, gc.getNumSpecialistInfos(), 'SPECIALIST_MERCHANT')

    iTotalSpecialistCount = (pCity.getSpecialistCount(iMerchant) + pCity.getFreeSpecialistCount(iMerchant))

    pCity.changeBonusGoodHappiness((iTotalSpecialistCount * iChange))

    return

def doSpicesCallback(argsList):
```

The **iChange** parameter passed to the python callback methods will either be **1** when the item is activated or **-1** when its deactivated

By multiplying the happiness values by iChange, we can correctly add or subtract the correct values when the items are activated or deactivated

```
    pCity = argsList[0]
    iBonus = argsList[1]
    iChange = argsList[2]

    iMerchant = CvUtil.findInfoTypeNum(gc.getSpecialistInfo, gc.getNumSpecialistInfos(), 'SPECIALIST_MERCHANT')

    iTotalSpecialistCount = (pCity.getSpecialistCount(iMerchant) + pCity.getFreeSpecialistCount(iMerchant))

    pCity.changeBonusGoodHappiness((iTotalSpecialistCount * iChange))

    return
```

And that's it. Cities will now get +1 Happiness from Furs and Spices Per Merchant in the city.

But there's still something missing. We've modded the game to allow a cool new feature, but players need to be able to know about it. To do this, we have to modify one of two possible tags in the Merchant entry of the CIV4SpecialistInfos.xml file.

We can do this:

```
<Help>TXT_KEY_SPECIALIST_MERCHANT_HELP</Help>
```

or this:

```
<PythonHelp>getMerchantHelp</PythonHelp>
```

The python help is pretty much only necessary when you have dynamic content that you want to display.

In our gametext file, we're going to add the following:

```
<Text>
        <Tag>TXT_KEY_SPECIALIST_MERCHANT_HELP</Tag>
        <English>[ICON_BULLET]+1 [ICON_HAPPY] with [LINK=BONUS_FUR]Furs[\LINK],
[LINK=BONUS_SPICES]Spices[\LINK]</English>
</Text>
```

Explanation: If you were a bit confused by all that, I'll help break it down now. Every time a Merchant specialist is added a city, the game will call the doMerchantCallback method. The method will check if the city has Furs and Spices and will add 1 extra happiness for each. The other two methods will be called when the city processes Spices or Furs (i.e. whenever the player connects either of these two resources to the city). This is precisely how the game does this for the Marketplace inside the DLL, so we're simply doing the same thing from python.

Tutorial 2: The Wonder

If you want to try this yourself, you can either modify an existing wonder, or create a new one. For the purposes of the this tutorial, I'm going to assume that we've created a wonder called King Richard's Crusade, which we want to spawn a new Maceman in the city in which it was built every 10 turns.

CvPythonDLLInterface.py:

```
def doKingRichardsCrusadeDoTurn(argsList):
    pCity = argsList[0]
```

```
    player = pCity.getOwner()
    iTurns = gc.getGame().getGameTurn()

    iUnitType = CvUtil.findInfoTypeNum(gc.getUnitInfo, gc.getNumUnitInfos(), 'UNIT_MACEMAN')

    if (iTurns % 10 == 0):
        player.initUnit(iUnitType, pCity.getX(), pCity.getY(), UnitAITypes.UNITAI_ATTACK_CITY, DirectionTypes.DIRECTION_SOUTH)
        szBuffer = localText.getText("TXT_KEY_MISC_JOINED_CRUSADE", (gc.getUnitInfo(iUnitType).getTextKey(), city.getNameKey()))

        CyInterface().addMessage(player.getID(), false, gc.getEVENT_MESSAGE_TIME(), szBuffer, "AS2D_DISCOVERBONUS",
InterfaceMessageTypes.MESSAGE_TYPE_MINOR_EVENT, gc.getUnitInfo(iUnitType).getButton(),
gc.getInfoTypeForString("COLOR_WHITE"), pCity.getX(), pCity.getY(), true, true)

    return
```

In the King Richard's Crusade entry in CIV4BuildingInfos.xml, modify this tag:

CIV4BuildingInfos.xml:

```
<Help>TXT_KEY_BUILDING_KING_RICHARDS_CRUSADE_HELP</Help>
```

In our gametext file, we're going to add the following:

```
<Text>
        <Tag>TXT_KEY_BUILDING_KING_RICHARDS_CRUSADE_HELP</Tag>
        <English>[ICON_BULLET]Creates a [LINK=UNIT_MACEMAN]Maceman[\LINK] in the City every 10
turns</English>
</Text>
```

Next we add the text that will appear on the screen each time a new Maceman is spawned in thsi city.

```
<Text>
        <Tag>TXT_KEY_MISC_JOINED_CRUSADE</Tag>
        <English>Religious zealots have formed a new &s1_Unit unit in %s2_City.</English>
</Text>
```

Tutorial 3: Pacifism

CvPythonDLLInterface.py:

```
def doPacifismCallback(argsList):
    iPlayer = argsList[0]
    iCivic = argsList[1]
    iChange = argsList[2]

    player = gc.getPlayer(iPlayer)

    player.changeStateReligionBuildingCommerce(CommerceTypes.COMMERCE_CULTURE, (2 * iChange))

    return
```

In the Pacifism entry in **CIV4BuildingInfos.xml**, modify this tag:

```
<PythonHelp>getPacifismHelp</PythonHelp>
```

In our gametext file, we're going to add the following:

```
<Text>
	<Tag>TXT_KEY_CIVIC_PACIFISM_HELP_1</Tag>
	<English>[ICON_BULLET]+2 %F1_Commerce from All State [ICON_RELIGION] Buildings</English>
</Text>
<Text>
	<Tag>TXT_KEY_CIVIC_PACIFISM_HELP_2</Tag>
	<English>[ICON_BULLET]+2 %F1_Commerce from All %F2_Religion Buildings</English>
</Text>
```

CvPythonDLLInterface.py:

```
def getPacifismHelp(argsList):
	iCivic = argsList[0]
	bCivilopedia = argsList[1]

	iPlayer = gc.getGame().getActivePlayer()

	player = gc.getPlayer(iPlayer)

	if (iPlayer != -1 and player.getStateReligion() != -1):
		szBuffer = localText.getText("TXT_KEY_CIVIC_PACIFISM_HELP_2",
(gc.getCommerceInfo(CommerceTypes.COMMERCE_CULTURE).getChar(), gc.getReligion(player.getStateReligion()).getChar()))
	else:
		szBuffer = localText.getText("TXT_KEY_CIVIC_PACIFISM_HELP_1",
(gc.getCommerceInfo(CommerceTypes.COMMERCE_CULTURE).getChar(), ))

	return szBuffer
```

Explanation: In this case, the code to actually make this work is fairly simple. It's the method for the help that is a bit trickier. Have you ever noticed that for certain things that pertain to a state religion, sometimes it reads "State [religion icon]" and sometimes it has the icon of your specific state religion? Code similar to that above in the SDK is what causes a specific religion icon to appear when looking at the item details when you have a state religion. So we've done the same thing here, so that our spiffy new bonus for Pacifism conforms to the style of help info elsewhere in the game.

Tutorial 4: New Build Type

Early armies often constructed their siege weaponry on the go rather than trying to move heavy, wheeled structures many hundreds of miles from their home country to the city they were besieging. We might want to represent this by creating a new Build type that workers can perform to build Catapults by chopping forests.

To start, we need to make a few changes.

First, we need to add a new entry for our new Build type.

CIV4BuildInfos.xml:

```
<BuildInfo>
	<Type>BUILD_CATAPULT</Type>
	<Description>TXT_KEY_BUILD_CATAPULT</Description>
	<Help>TXT_KEY_BUILD_CATAPULT_HELP</Help>
	<PrereqTech>TECH_CONSTRUCTION</PrereqTech>
	<iTime>0</iTime>
	<iCost>0</iCost>
```

```
		<bKill>0</bKill>
		<ImprovementType>NONE</ImprovementType>
		<RouteType>NONE</RouteType>
		<EntityEvent>ENTITY_EVENT_CHOP</EntityEvent>
		<FeatureStructs>
			<FeatureStruct>
				<FeatureType>FEATURE_FOREST</FeatureType>
				<PrereqTech>TECH_BRONZE_WORKING</PrereqTech>
				<iTime>300</iTime>
				<iProduction>0</iProduction>
				<bRemove>1</bRemove>
			</FeatureStruct>
		</FeatureStructs>
		<HotKey>KB_C</HotKey>
		<bAltDown>0</bAltDown>
		<bShiftDown>1</bShiftDown>
		<bCtrlDown>1</bCtrlDown>
		<iHotKeyPriority>0</iHotKeyPriority>

<Button>,Art/Interface/Buttons/Builds/BuildChopDown.dds,Art/Interface/Buttons/Actions_Builds_LeaderHeads_Specialists_Atlas.dds,7,8</Button>
</BuildInfo>
```

We also need to go into the Worker entry in **CIV4UnitInfos.xml** and add the following to the list of Build types:

```
		<Build>
			<BuildType>BUILD_CATAPULT</BuildType>
			<bBuild>1</bBuild>
		</Build>
```

In **PythonCallbackDefines.xml**, we need to change the USE_CAN_BUILD_CALLBACK entry from a 0 to a 1. This will allow us to run a check to see if this Build can be done on this tile.

CvGameUtil.py:

```
def canBuild(self, argsList):
	iX, iY, iBuild, iPlayer = argsList

	plot = gc.getMap().plot(iX, iY)

	iForest = CvUtil.findInfoTypeNum(gc.getFeatureInfo,gc.getNumFeatureInfos(),'FEATURE_FOREST')

	iBuildCatapult = CvUtil.findInfoTypeNum(gc.getBuildInfo,gc.getNumBuildInfos(),'BUILD_CATAPULT')

	if (plot.getFeatureType() != iForest and iBuild == iBuildCatapult):
		return False

	return True
```

We only want this Build to be performed on tiles with Forests, so we use this pre-existing python method to determine if this plot is a valid place to build a Catapult.

Then we must write the code for what will happen once the Build has been completed.

CvPythonDLLInterface.py:

```
def doBuildCatapultCallback(argsList):
	iX = argsList[0]
	iY = argsList[1]
```

```
		iTeam = argsList[2]

		team = gc.getTeam(iTeam)
		player = gc.getPlayer(team.getLeaderID())

		iWorker = CvUtil.findInfoTypeNum(gc.getUnitInfo, gc.getNumUnitInfos(), 'UNIT_WORKER')
		iUnitType = CvUtil.findInfoTypeNum(gc.getUnitInfo, gc.getNumUnitInfos(), 'UNIT_CATAPULT')

		player.initUnit(iUnitType, iX, iY, UnitAITypes.UNITAI_ATTACK_CITY, DirectionTypes.DIRECTION_SOUTH)
		szBuffer = localText.getText("TXT_KEY_MISC_BUILT_CATAPULT", (gc.getUnitInfo(iWorker).getTextKey(),
gc.getUnitInfo(iUnitType).getTextKey()))

		CyInterface().addMessage(player.getID(), false, gc.getEVENT_MESSAGE_TIME(), szBuffer, "AS2D_DISCOVERBONUS",
InterfaceMessageTypes.MESSAGE_TYPE_MINOR_EVENT, gc.getUnitInfo(iUnitType).getButton(), gc.getInfoTypeForString("COLOR_WHITE"), iX,
iY, true, true)
```

Last, we have to add the text in our gametext file that's to be displayed when the build is complete and the unit is constructed.

```
<Text>
		<Tag>TXT_KEY_MISC_BUILT_CATAPULT</Tag>
		<English>Our %s1_Worker has constructed a %s2_Catapult</English>
</Text>
<Text>
		<Tag>TXT_KEY_BUILD_CATAPULT_HELP</Tag>
		<English>Construct a Catapult on this tile</English>
</Text>
```

Explanation: The FeatureStructs tag in the Build entry in CIV4BuildInfos.xml controls whether the underlying Forest will be cleared and in what amount of time. If we wanted to make it so that the forests aren't cleared when we build our catapult, we would just change to <bRemove>1</bRemove> to <bRemove>0</bRemove>.

In order for the canBuild function to be called from the DLL, the USE_CAN_BUILD_CALLBACK has to be set to true in the PythonCallbackDefines.xml. Any entry in this file with a value of 0 will be ignored, and the python method will not be called.

The doBuildCatapultCallback method is called right when the build finishes and will initiate our new Catapult unit as well as inform its owner that the task has been completed.

Tutorial 6: Corpse Chucking

In the old days of siege warfare when you could pretty much hurl anything you wanted over an enemy's city walls, it was not uncommon to throw corpses of dead enemy soldiers and infected animals over the city walls to destroy morale and spread disease in the city walls. Let's give Catapults and Trebuchets a new ability to spread unhappiness and unhealthiness in cities in this same way.

CIV4ActionButtonInfos.xml:

```
<ActionButtonInfo>
		<Type>ACTIONBUTTON_SPREAD_DISEASE</Type>
		<Description>TXT_KEY_ACTIONBUTTON_SPREAD_DISEASE</Description>
		<Civilopedia>TXT_KEY_PEDIA_ACTIONBUTTON_SPREAD_DISEASE</Civilopedia>
		<Strategy>TXT_KEY_STRATEGY_ACTIONBUTTON_SPREAD_DISEASE</Strategy>
		<Help>TXT_KEY_ACTIONBUTTON_SPREAD_DISEASE_HELP</Help>
		<EntityEventMission>MISSION_BOMBARD</EntityEventMission>
		<EffectType>EFFECT_CITY_DISEASED</EffectType>
```

```
        <bPickPlot>1</bPickPlot>
        <iRange>1</iRange>
        <PythonCanDo>canSpreadDisease</PythonCanDo>
        <PythonCallback/>
        <PythonCanDoAtPlot>canSpreadDiseaseAtPlot</PythonCanDoAtPlot>
        <PythonCallbackAtPlot>doSpreadDiseaseAtPlot</PythonCallbackAtPlot>
        <PythonHelp/>
        <HotKey></HotKey>
        <bAltDown>0</bAltDown>
        <bShiftDown>0</bShiftDown>
        <bCtrlDown>0</bCtrlDown>
        <iHotKeyPriority>0</iHotKeyPriority>
        <Sound/>

<Button>,Art/Interface/Buttons/Builds/BuildRoad.dds,Art/Interface/Buttons/Python_Atlas.dds,1,8</Button>
</ActionButtonInfo>
```

While we could do this several ways, we want ranged attack here, so **bPickPlot** must be true.

iRange is **1** because we do not want the siege weapons attacking farther than 1 tile away.

There's a couple of different ways we could do this. We actually don't have to make this a ranged attack that requires the player to pick a plot to attack. The regular City Bombardment does not require the player to pick a plot to bombard. But for the sake of this tutorial, we are going to set bPickPlot to 1. We will set iRange to 1 because we don't want Catapults and Trebuchet to be able to launch this attack farther than an adjacent plot.

Okay, so what have we done so far? We've created the button we want and told the game where to grab the graphic for it. See the image below to see the button in the game with the graphic I have chosen (I apologize in advance).

Image 1:

We need to create three python methods. One called ***canSpreadDisease***, will determine whether or not the action can be perform at all by this unit. If not, the button will not appear. The next is ***canSpreadDiseaseAtPlot***. This will be called

when the player clicks the button and has to choose a tile to attack. The method ***doSpreadDiseaseAtPlot*** will actually make the changes caused by attacking this plot.

CvPythonDLLInterface.py:

```
def canSpreadDisease(argsList):
	pUnit = argsList[0]

	if (pUnit.getUnitType() == CvUtil.findInfoTypeNum(gc.getUnitInfo, gc.getNumUnitInfos(), 'UNIT_CATAPULT')):
		return True
	if (pUnit.getUnitType() == CvUtil.findInfoTypeNum(gc.getUnitInfo, gc.getNumUnitInfos(), 'UNIT_TREBUCHET')):
		return True
	return False

def canSpreadDiseaseAtPlot(argsList):
	pUnit = argsList[0]
	iActionButton = argsList[1]
	iX = argsList[2]
	iY = argsList[3]

	map = gc.getMap()
	pPlot = map.plot(iX, iY)

	if (pPlot.isEnemyCity(pUnit)):
		pCity = pPlot.getPlotCity()
		if ((pCity.getEspionageHealthCounter() < 8) and (pCity.getEspionageHappinessCount()  < 8)):
			return True

	return False

def doSpreadDiseaseAtPlot(argsList):
	pUnit = argsList[0]
	iActionButton = argsList[1]
	iX = argsList[2]
	iY = argsList[3]

	map = gc.getMap()
	pPlot = map.plot(iX, iY)

	pCity = pPlot.getPlotCity()

	if (not pCity.isNone()):
		iChange = -(pCity.getEspionageHealthCounter() - 8)

		if (iChange < 2):
			pCity.changeEspionageHealthCounter(iChange)
		else:
			pCity.changeEspionageHealthCounter(2)

		iChange = –(pCity.getEspionageHappinessCounter() - 8)

		if (iChange < 2):
			pCity.changeEspionageHappinessCounter(iChange)
		else:
			pCity.changeEspionageHappinessCounter(2)
		pUnit.setMadeAttack(1)
		pUnit.changeMoves(gc.getMOVE_DENOMINATOR())

		szBuffer = localText.getText("TXT_KEY_MISC_OURS_SPREAD_DISEASE", (pUnit.getNameKey(),
pCity.getNameKey()))
		CyInterface().addMessage(pUnit.getOwner(), false, gc.getEVENT_MESSAGE_TIME(), szBuffer,
"AS2D_DISCOVERBONUS", InterfaceMessageTypes.MESSAGE_TYPE_MINOR_EVENT, gc.getUnitInfo(pUnit.getUnitType()).getButton(),
gc.getInfoTypeForString("COLOR_GREEN"), iX, iY, true, true)

		szBuffer = localText.getText("TXT_KEY_MISC_THEIRS_SPREAD_DISEASE", (pUnit.getNameKey(),
pCity.getNameKey()))
		CyInterface().addMessage(pCity.getOwner(), false, gc.getEVENT_MESSAGE_TIME(), szBuffer,
"AS2D_DISCOVERBONUS", InterfaceMessageTypes.MESSAGE_TYPE_MINOR_EVENT, gc.getUnitInfo(pUnit.getUnitType()).getButton(),
gc.getInfoTypeForString("COLOR_RED"), iX, iY, true, true)
```

The above code finds the city located on the plot and then uses the espionage happiness and health counters to create negative effects on the city being bombarded. The more siege weapons using the attack on the city, the worse the effect.

The last few lines of code have a message appear for both the attacker and the defender informing them of the attack.

Now that we've added the code to actually do something, we need to add the text that will appear when the attack occurs. In our gametext file, we add:

```
<TEXT>
        <Tag> TXT_KEY_MISC_OURS_SPREAD_DISEASE</Tag>
        <English>Our %s1_Unit has spread disease and weakened morale in %s2_City</English>
</TEXT>
<TEXT>
        <Tag> TXT_KEY_MISC_THEIRS_SPREAD_DISEASE</Tag>
        <English>An enemy %s1_Unit has spread disease and weakened morale in %s2_City</English>
</TEXT>
```

Image 2:

Image 3:


11:31 AM
Moscow: 4
Defense: +40%
5 - 12
STARVATION!
6 < 10 = -4
11
5 < 10 = +4
Note that because we're using the espionage code, the help text is espionage-specific. You might want to change the text in your own mod to something more generic.