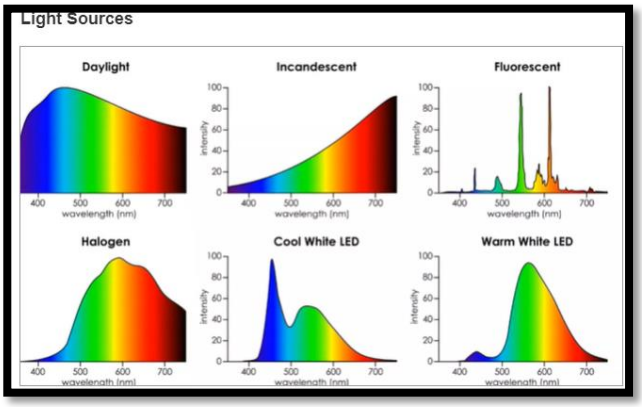
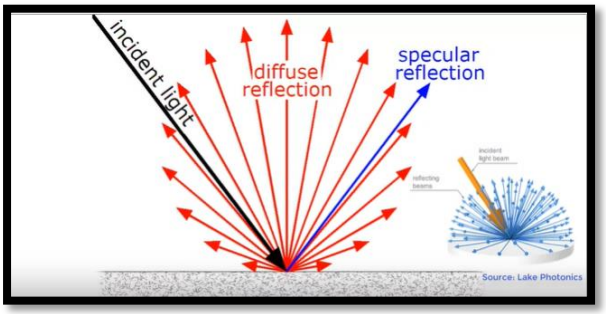
DAILY ASSESSMENT

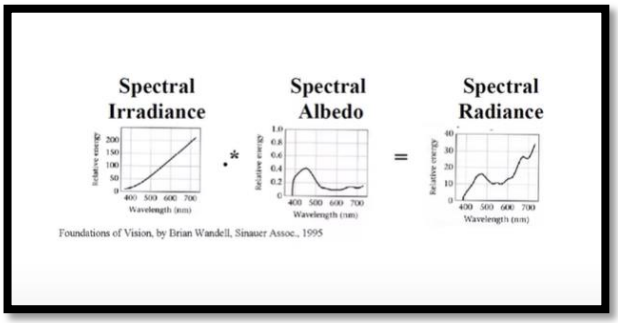
Date:	**14/07/2020**	**Name:**	**Davis S. Patel**
Course:	**Computer Vision Basics**	**USN:**	**4AL16EC045**
Topic:	**Week 2**	**Semester & Section:**	**8th - A**
GitHub Repository:	**Davis**		

FORENOON SESSION DETAILS

Image of session







REPORT – Resources: Color, Light, & Image Formation

To get the right effect from your lighting, it pays to know how different sizes and shapes of light sources affect how people, objects and surfaces will appear when illuminated. An important step is to know whether the light source you want to use is a point source, linear source or area source, which provide a palette of choices ranging from a high level of intensity and shadowing for point sources to very little intensity and shadowing for area sources.

Point sources are small lamps (light bulbs), often using a clear outer glass bulb that reveals the arc tube or bare incandescent filament. Incandescent, halogen and LED lamps are typically point sources. The light produced by point sources is generally intense, and more likely to produce pronounced shadows. As a result, point sources are ideally suited to accent lighting—individually highlighting art and other focal points in a space—and any application where we want to accentuate detail through shadow, such as grazing a brick wall to visually enrich its texture.

Linear sources emit diffuse output from the surface of the lamp, softening shadows. Tubular and compact fluorescent lamps are examples of linear sources. These lamps can be very effective for ambient lighting in work areas, cove lighting and wall washing for large surfaces.

Area sources are not electric lamps but instead large non-specular surfaces that reflect or emit diffused light output, such as a ceiling reflecting light from an indirect luminaire (light fixture), or the surface of a luminous bowl pendant luminaire. When washing a light-color wall with light, for example, the wall itself will become a light source in the space, reflecting light from the wall to other parts of the space.

Man has senses to observe the world around him. Computer Vision is actually the better version of our human senses. Computer Vision is capable of making a deep analysis of images or a series of them in just a few seconds. This is one of the most complex processes which we ever attempted to comprehend. Inventing a machine that owns senses which are even better than our own, is something magical. The most impressive thing is, that we actually do not exactly know by what kind of complex process we created this kind of super creatures (BMVA, 2017).

Computer vision can analyze, is able to extract automatically, and will understand useful information from just a glimpse of a single image or a sequence of images. This mechanism is based on a theoretical framework of trained algorithms. This process works roughly the same as by humans. The algorithms are trained to go quick through a classification of images, objects, colors, sizes etc., this takes place in a tiny fraction of a second. This classification is similar to our own visual cortex, which contains a framework of references to things “we already know.” The path that the algorithm is going to follow is actually already determined, but it contains an infinite number of possibilities. There is almost no conscious effort, because it has been programmed precisely.

It is striking that the cameras of computer vision devices are not that much better than simple 19th century pinhole cameras. Computer vision is all about the “mean shift algorithm” which is the so-called “Camshift” that forms the mediator programmed on robust statistics and operates on probability distributions (Brodsky, 2). The movement amount of the camera has to be very sensitive to the frame rate in order to recognize complex patterns. Computer-rendered graphics or game scenes with simple views (like a blue sky) are rendered much faster than complex views (like cities). It is important that the final rate movement should not depend on the complexity of a particular 3D view, to overcome this, they use empirical observations (Bradsky, 8). Sets of neurons excite one another in contrasts, the higher level network aggregate these patterns into meta-patterns: this complementary process creates the image with the required descriptions. The development process of computer vision is a collaboration between computer scientists, engineers, psychologists, neuroscientists and philosophers, who jointly determine the working definition of our mind.

Computer vision is implemented today in self-driving cars, it is in factory robots, and in your personal smart phone. But the possibilities of these devices are limited. Labsix, a group of MIT students recently published a research paper. They have developed an algorithm that is fixed to deceive image classifiers, from these “errors” they develop new programming’s to optimize computer vision. They analyze how the system makes decisions. Common images are sabotaged by the “contradictory” algorithm because the pixels are minimally changed. The algorithm preserves exactly the right combination of sabotaged pixels during the process, but the small

changes in the pixels cannot be read by the system. There are plenty of research tests to counter adversarial examples, computer vision will not be trusted until adversarial attacks are impossible, or at least hard to pull of (Snow, 2017).

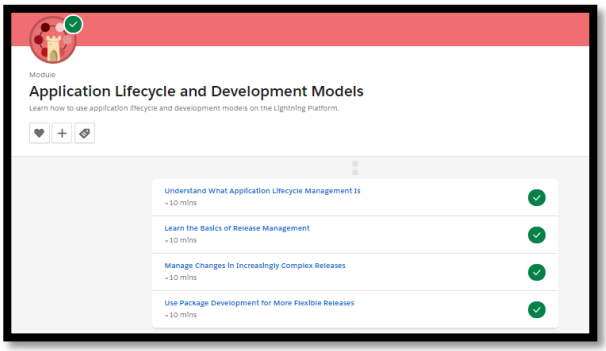
In the visual arts, color theory is a body of practical guidance to color mixing and the visual effects of a specific color combination. There are also definitions (or categories) of colors based on the color wheel: primary color, secondary color, and tertiary color. Although color theory principles first appeared in the writings of Leone Battista Alberti (c. 1435) and the notebooks of Leonardo da Vinci (c. 1490), a tradition of "colory theory" began in the 18th century, initially within a partisan controversy over Isaac Newton's theory of color (*Opticks*, 1704) and the nature of primary colors. From there it developed as an independent artistic tradition with only superficial reference to colorimetry and vision science.

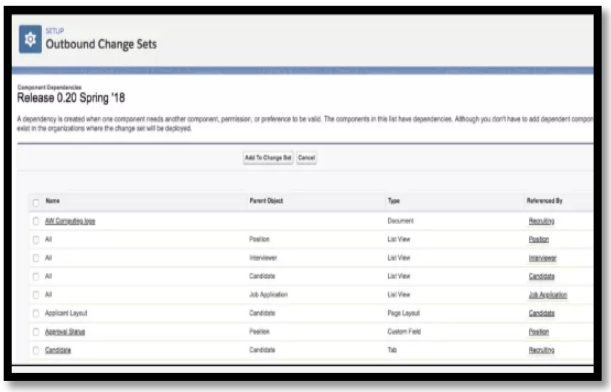
DAILY ASSESSMENT

Date:	**14/07/2020**	**Name:**	**Davis S. Patel**
Course:	**Salesforce Developer**	**USN:**	**4AL16EC045**
Topic:	**Application Lifecycle and Development Models**	**Semester & Section:**	**8th - A**
GitHub Repository:	**Davis**		

AFTERNOON SESSION DETAILS

Image of Session





REPORT –

Salesforce provides various development tools and processes to meet the needs of customers. This module introduces the application lifecycle management (ALM) process and the three development models.

- Change set development
- Org development
- Package development

At a high level, all three development models follow the same ALM process. But the models differ in the way that they let we manage changes to your org. Controlling change is a big deal in software development, and we can choose the development model that best suits our situation if you understand our option.

Application lifecycle management

ALM provides them with process and policies that help them build apps *smoothly* and therefore faster, without breaking things. Apps and tools can vary, but the steps in the ALM cycle apply to any Salesforce development project.

Step 1: Plan Release

Start your customization or development project with a plan. Gather requirements and analyze them. Have your product manager (or equivalent) create design specifications and share them with the development team for implementation. Determine the various development and testing environments the team needs as the project progresses through the ALM cycle.

Step 2: Develop

Complete the work, following the design specifications. Perform the work in an environment containing a copy of the production org's metadata, but with no production data. Develop on Lightning Platform using an appropriate combination of declarative tools (Process Builder, the

Custom Object wizard, and others in the UI) and programmatic tools (Developer Console, Source Code Editor, or Visual Studio Code).

Step 3: Test

Exercise the changes you're making to check that they work as intended before you integrate them with other people's work. Do your testing in the same type of environment as you used in the develop step, but keep your development and integrated testing environments separate. At this point, focus on testing your changes themselves, not on understanding how your changes affect other parts of the release or the app as a whole.

Step 4: Build Release

Aggregate all the assets you created or modified during the develop stage into a single release artifact: a logical bundle of customizations that you deploy to production. From this point on, focus on what you're going to release, not on the contributions of individuals.

Step 5: Test Release

Test what you're actually going to deploy, but test safely in a staging environment that mimics production as much as possible. Use a realistic amount of representative production data. Connect your test environments with all the external systems they need to mimic your production system's integration points. Run full regression and final performance tests in this step. Test the release with a small set of experienced people who provide feedback (a technique called user-acceptance testing).

Step 6: Release

When you've completed your testing and met your quality benchmarks, you can deploy the customization to production. Train your employees and partners so they understand the changes. If a release has significant user impact, create a separate environment with realistic data for training users.

In package development, you manage different customizations as separate packages, not as one big release of changes to the org. Remember how in change set development you manage a set of changes from multiple projects as though they're going into one container? When releases become so complex that it makes sense to manage the org as multiple containers, it's time to move to the package development model. If your team is already building modular release artifacts on other platforms, they'll find some similarities working in package development.

For example, a package might contain any of the following customizations.

- Custom Force.com apps they built in-house
- Extensions of Sales Cloud, Service Cloud, and so on
- Extensions of an AppExchange app
- Updates to shared libraries and functionality

When you're working in a package development model, you build a release artifact you can test and release independently from artifacts for other projects. Instead of a set of changes relative to production, your team creates a package that contains all the relevant metadata. The metadata in the package includes both changed and unchanged components.

Organizing metadata updates by packages also helps you create a better mental model of how the metadata in your org is structured. If you want to manage your org as multiple containers, each package represents one of those containers.

A package version is a fixed snapshot of the package contents and related metadata. The package version lets you manage what's different each time you release or deploy a specific set of changes added to a package. If you're introducing metadata changes to an already deployed package, you upgrade from the current package version to the newer package version.

- Creating a clear audit trail of how org metadata changed over time
- Increasing productivity by freeing up time currently spent on tracking Setup changes
- Gaining greater release flexibility, because each package can be developed, tested, and released independently of packages for other projects

There's a development environment available in package development called scratch orgs. Scratch orgs play a key role in significantly reducing the changes Calvin and his team need to track for each release.

Scratch orgs are empty orgs (no metadata or data) that are easy to create and dispose of as needed. You can configure scratch orgs to be different Salesforce editions with different features and preferences. What's more, you can re-use and share the scratch org definition file with other team members, because it's part of the project integrated into the VCS. This way, it's much simpler for you all to work in the same org configuration while each having your own development environment.

When you use scratch orgs for development, you first push the source from your project in the VCS to sync the scratch org with the same metadata. If you are planning to use Setup for development, the changes you make are tracked automatically. You can pull down the modifications you made to include them in your project and use your VCS to commit all the changes.